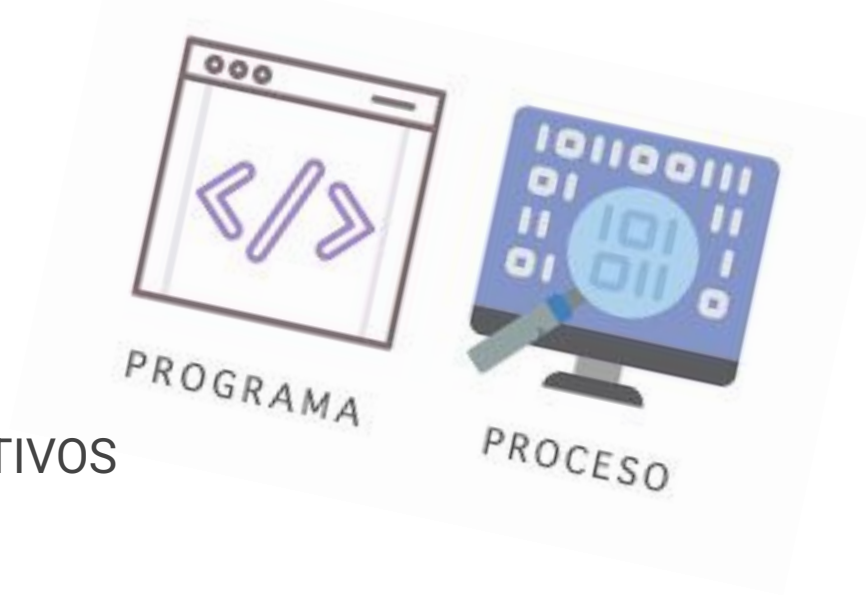


Unidad 2

GESTIÓN DE PROCESOS



Cátedra: ARQUITECTURA Y SISTEMAS OPERATIVOS

Docente: Ing. Guillermo Andrés Martínez

e-mail: gamartinez@udc.edu.ar

PROCESOS - Definición y Concepto

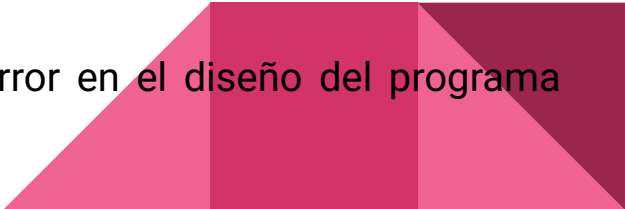
Un proceso es una instancia en ejecución de un programa.

Para entenderlo mejor, podemos desglosar esta definición:

- **Programa:** Es un conjunto de instrucciones estáticas, código escrito que define una tarea a realizar. Un programa por sí solo no hace nada, es como una receta escrita.
- **Instancia:** Es una copia del programa en memoria, lista para ser ejecutada. Es como tener la receta en la mano y los ingredientes preparados.
- **En ejecución:** Significa que el programa está activo, el procesador está llevando a cabo las instrucciones. Es como estar cocinando siguiendo la receta.

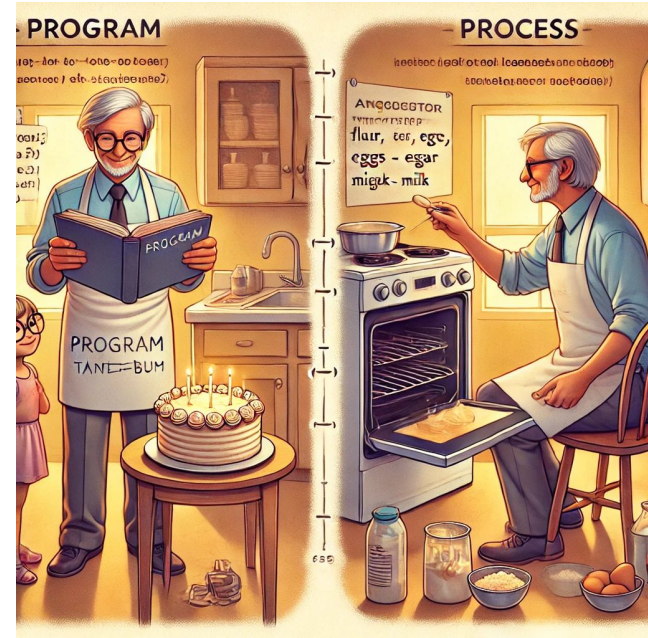


PROCESOS - Definición y Concepto

- Los procesos en Linux son ni más ni menos que programas que están corriendo en un momento dado.
 - En nuestro sistema operativo el árbol de procesos está representado en el directorio /proc
 - Cuando un proceso A inicia otro proceso B, se dice que A es el padre de B (o lo que es lo mismo, que B es hijo de A).
 - Los procesos normales generalmente son ejecutados en una terminal y corren en el sistema operativo a nombre de un usuario.
 - Los procesos daemon también se ejecutan a nombre de un usuario pero no tienen salida directa por una terminal, sino que corren en segundo plano. Generalmente los conocemos como servicios, y en vez de utilizar una terminal para escuchar por un requerimiento, lo hacen a través de un puerto.
 - Los procesos zombies son aquellos que han completado su ejecución pero aún tienen una entrada en la tabla de procesos debido a que no han enviado la señal de finalización a su proceso padre (o este último no ha podido leerla).
 - Usualmente la presencia de este tipo de procesos indica un error en el diseño del programa involucrado.
- 

Analogía

Para entender lo que es un proceso y la diferencia entre un programa y un proceso, A. S. Tanenbaum propone la analogía "Un científico computacional con mente culinaria hornea un pastel de cumpleaños para su hija; tiene la receta para un pastel de cumpleaños y una cocina bien equipada con todos los ingredientes necesarios, harina, huevo, azúcar, leche, etcétera." Situando cada parte de la analogía se puede decir que la receta representa el programa (el algoritmo), el científico computacional es el procesador y los ingredientes son las entradas del programa. El proceso es la actividad que consiste en que el científico computacional vaya leyendo la receta, obteniendo los ingredientes y horneando el pastel.



Características de un Proceso

- **Espacio de direcciones propio:** Cada proceso tiene su propia área de memoria donde se cargan el código, los datos y la pila. Esto aísla los procesos entre sí, evitando que interfieran.
- **Recursos asignados:** El sistema operativo asigna recursos al proceso, como tiempo de CPU, memoria, dispositivos de entrada/salida, etc.
- **Contexto:** El contexto de un proceso es el conjunto de valores de los registros del procesador y otros datos que permiten al sistema operativo pausar y reanudar la ejecución del proceso.



Estados y transiciones de los procesos

Nuevo→Listo: Al crearse un proceso pasa inmediatamente al estado nuevo

Listo→Ejecutando: En el estado de listo, el proceso solo espera para que se le asigne un procesador para ejecutar (tener en cuenta que puede existir más de un procesador en el sistema). Al liberarse un procesador el planificador (*scheduler*) selecciona el próximo proceso, según algún criterio definido, a ejecutar.

Ejecutando→Listo: Ante una interrupción que se genere, el proceso puede perder el recurso procesador y pasar al estado de listo. El planificador será el encargado de seleccionar el próximo proceso a ejecutar.



Estados y transiciones de los procesos

Ejecutando→Bloqueado: A medida que el proceso ejecuta instrucciones realiza pedidos en distintos componentes (ej.: genera un pedido de E/S). Teniendo en cuenta que el pedido puede demorar y, además, si está en un sistema multiprogramado, el proceso es puesto en una cola de espera hasta que se complete su pedido. De esta forma, se logra utilizar en forma más eficiente el procesador.

Bloqueado→Listo: Una vez que ocurre el evento que el proceso estaba esperando en la cola de espera, el proceso es puesto nuevamente en la cola de procesos listos.

Ejecutando→Terminado: Cuando el proceso ejecuta su última instrucción pasa al estado terminado. El sistema libera las estructuras que representan al proceso.



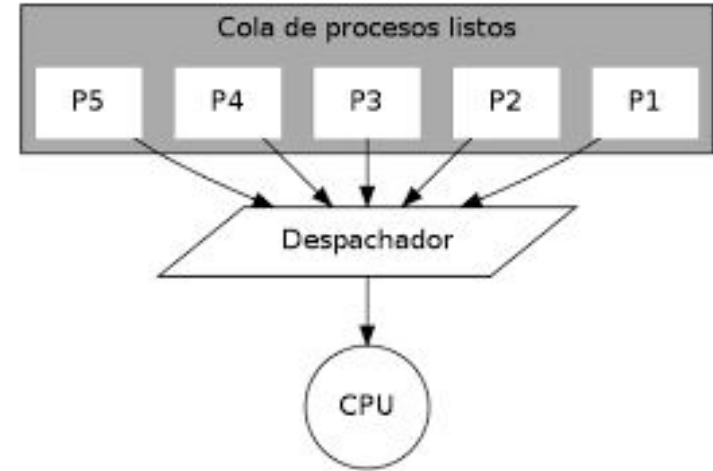
Despachador

OBJETIVO PRINCIPAL DEL DESPACHADOR:

Optimizar la eficiencia del sistema, asignando el procesador (núcleos) a los diferentes procesos utilizando criterios de planificación.

FUNCIONES

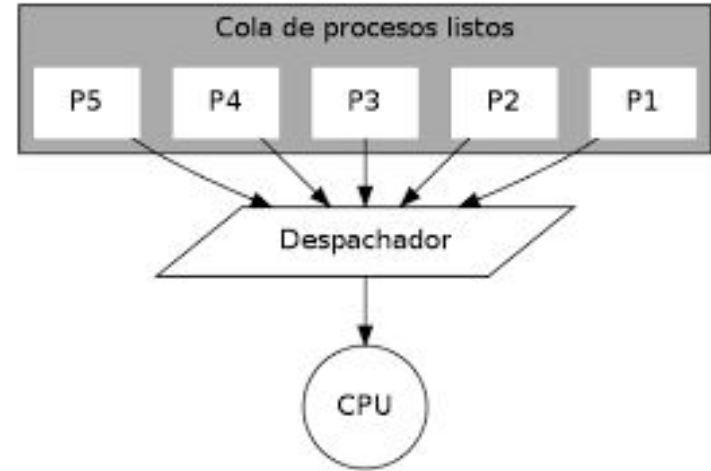
- El despachador examina la prioridad de los procesos.
- Controla los recursos de una computadora y los asigna entre los usuarios.
- Permite a los usuarios correr sus programas.
- Controla los dispositivos de periféricos conectados a la máquina.
- Cambio de contexto.
- Cambio a modo usuario.



Despachador

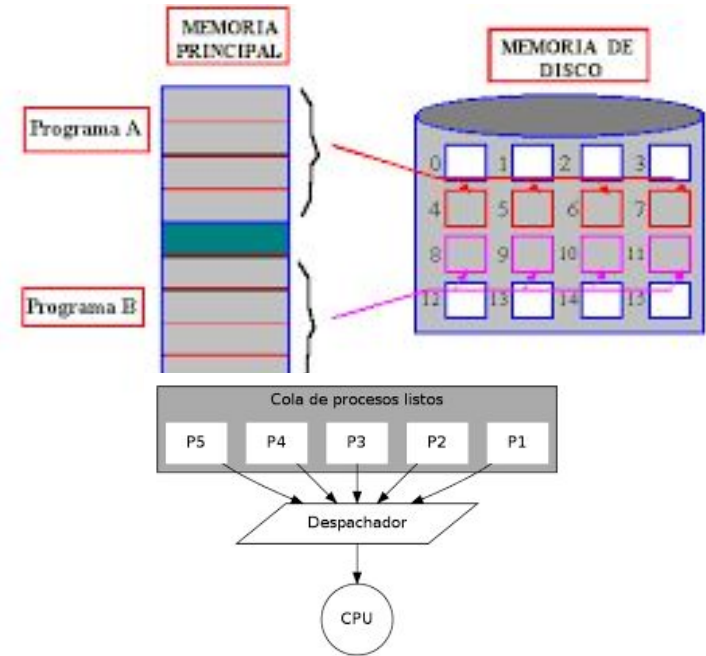
Es un módulo que entrega el control de la

- CPU
- Cambio de contexto
- Cambio a modo usuario
- Actualiza el PC
- Latencia de despacho



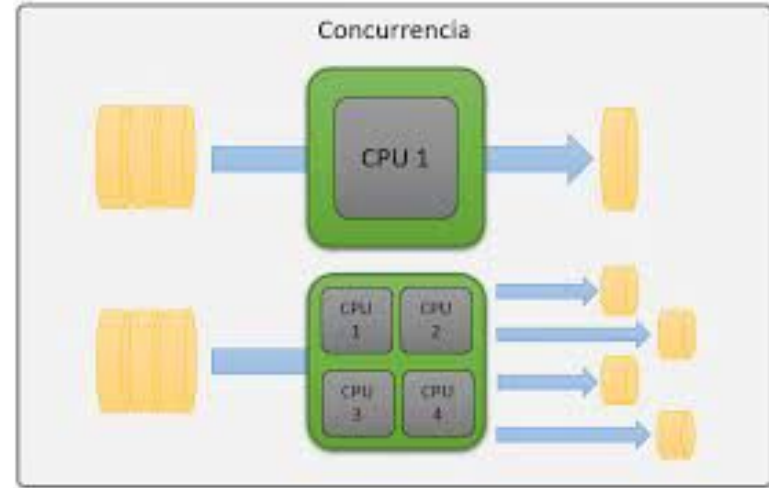
Tipos de Despachadores

- **De largo plazo:** Este tipo de despachador es invocado cada vez que un proceso termina y abandona el sistema. Se encarga de la transición de un proceso del estado de dormido al estado de listo.
- **De mediano plazo.** Cuando existen procesos que necesitan un uso intensivo de las facilidades de entrada y salida, y que por ello permanezcan suspendidos, puede ser que éstos procesos se quiten temporalmente de memoria principal y se guarden en memoria secundaria, hasta que su condición de espera haya concluido. El despachador de mediano plazo se encarga del manejo de procesos que temporalmente se han enviado a memoria secundaria. En términos del diagrama de transición de estados, el despachador de mediano plazo se encarga de la transición suspendido a listo.
- **De corto plazo.** El despachador de corto plazo asigna el CPU entre los procesos listos en memoria principal. Su objetivo principal es maximizar la eficiencia del sistema de acuerdo con ciertos criterios. Ya que se encarga de las transiciones de listo a ejecutándose. En la práctica, el despachador de corto plazo se invoca cada vez que ocurre un evento que modifique el estado global del sistema.



Concurrencia de procesos

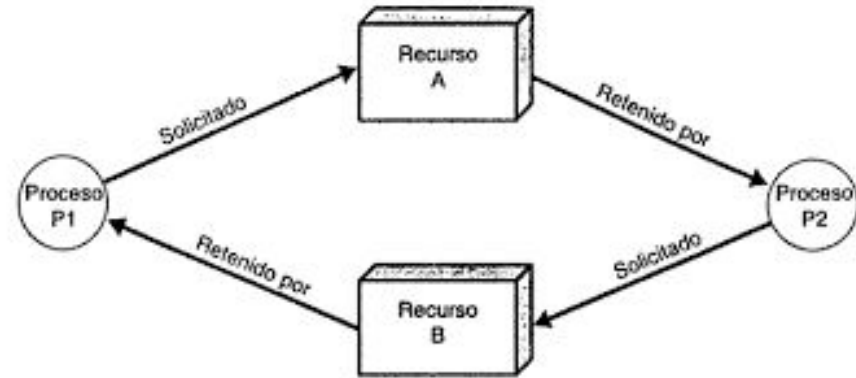
La concurrencia de procesos se refiere a las situaciones en las que dos o más procesos puedan coincidir en el acceso a un recurso compartido o, dicho de otra forma, que requieran coordinarse en su ejecución. Para evitar dicha coincidencia, el sistema operativo ofrece mecanismos de arbitraje que permiten coordinar la ejecución de los procesos.



Mecanismos de arbitraje

Los mecanismos de arbitraje son reglas que permiten asignar recursos de la máquina a más de un usuario o programa. También son mecanismos de sincronización que coordinan la ejecución de los procesos.

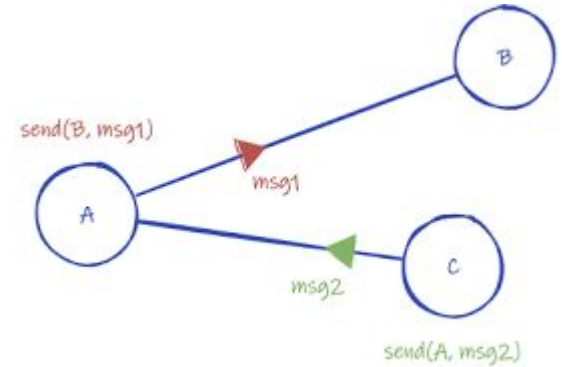
- Permiten que los procesos coordinen su ejecución
- Evitan que dos o más procesos accedan simultáneamente a un recurso que no se puede compartir
- Garantizan el acceso mutuamente exclusivo a un recurso compartido entre subprocesos



Mecanismos de arbitraje

Los mecanismos de arbitraje que ofrece el sistema operativo son:

- Mecanismos de **sincronización**: el sistema operativo ofrece mecanismos que permiten a los procesos coordinar su ejecución para conseguir el objetivo sin que sucedan situaciones no deseadas, como por ejemplo que dos o más procesos coincidan simultáneamente en el acceso a un cierto recurso que no se puede compartir.
- Mecanismos de **mensajería**: el sistema operativo ofrece mecanismos de comunicación entre procesos mediante mensajes. El intercambio de mensajes entre procesos permite coordinarlos.



Programación concurrente

El término programación concurrente se emplea con frecuencia para referirse a un conjunto de programas que funcionan en cooperación.

Hay tres formas de interacción entre procesos cooperativos:

- **Concurrencia:** Hay un recurso común, si varios procesos modificaran la misma información a la vez, cada uno podría destruir parte del trabajo de los demás. Si lo hacen uno tras otro, en serie, se obtendrá el resultado correcto.
- **Sincronización:** El Sistema Operativo se encarga de enviarle señales a los procesos para coordinar su evolución y conseguir que progrese armónicamente.
- **Comunicación:** El S.O. transmite mensajes entre los procesos, que se usan para intercambiar, enviar o recibir información.



Comunicación entre Procesos

- La comunicación entre procesos (**IPC**) es un mecanismo que permite que programas y procesos se comuniquen e intercambien datos. Es fundamental para el funcionamiento de los sistemas operativos y aplicaciones.
- Los procesos pueden comunicarse entre sí a través de compartir espacios de memoria, ya sean variables compartidas o buffers, o a través de las herramientas provistas por las rutinas de IPC.
- La comunicación se establece siguiendo una serie de reglas (protocolos de comunicación).



Comunicación entre Procesos

Existen 4 formas que usan los procesos para comunicarse:

1. **Mensajes**
2. **Archivos compartidos**
3. **Memoria Compartida**
4. **Socket (para el mismo equipo y remoto)**



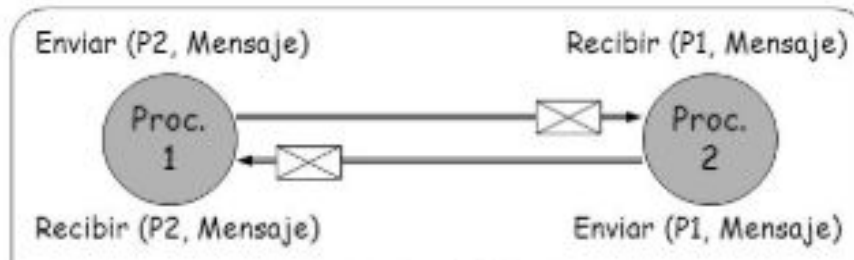
Comunicación entre Procesos - Paso de Mensajes

Una opción para que dos procesos se comuniquen es el paso de mensajes entre ellos. Un proceso P envía un mensaje a otro proceso Q (send). El proceso Q recibe dicho mensaje (receive). La comunicación puede ser:

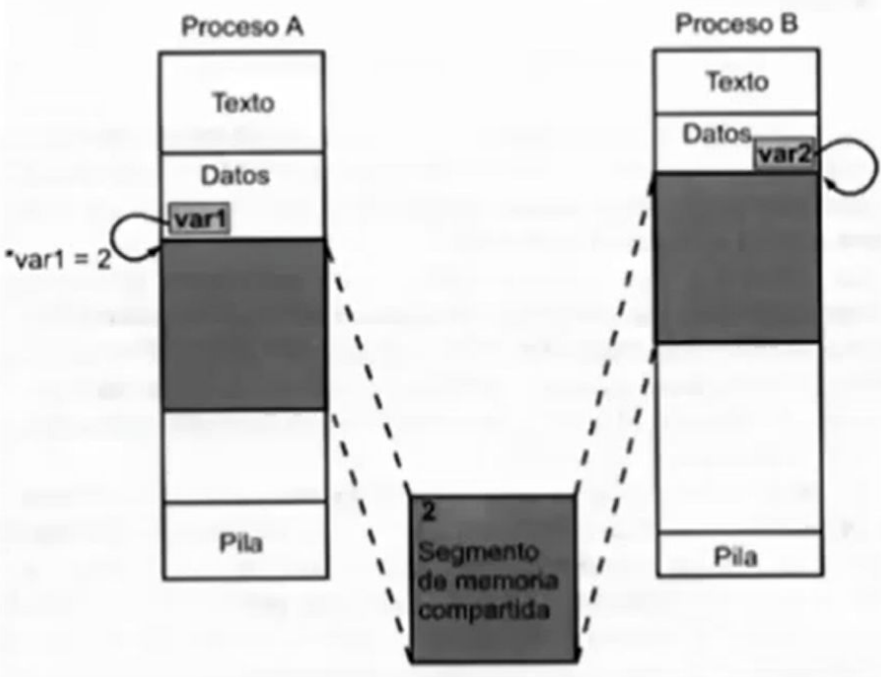
Pasar mensajes es útil para pequeños datos, pero es lento porque requiere una llamada al sistema para cada mensaje.

Modelos comunicación. Comunicación directa

- Cada proceso tiene su propio buzón.
- Cada proceso implicado debe indicar explícitamente el nombre del receptor o emisor.



Comunicación entre Procesos- Memoria compartida



Una opción para dos procesos que se comunican es compartir una región de memoria.

- Un proceso crea una región de memoria compartida en su heap asignándole ciertos permisos.
- Otros procesos se asocian a dicha región.
- Todos pueden usar dicha región según los permisos establecidos.

Comunicación entre Procesos- Socket

La comunicación entre procesos a través de socket a diferencia de las comunicaciones anteriores sirve para comunicarnos tanto con el mismo equipo como con un equipo remoto.

Un Socket contiene la dirección IP (protocolo IP) del equipo y el puerto (Protocolo TCP o UDP)

- La IP identifica un equipo en la RED
- Un Puerto identifica al proceso (servicio) dentro de la PC.



Figura 4.3. Comunicación cliente-servidor.

Para poder comunicarse usa la metodología cliente servidor, donde el cliente realiza peticiones al servidor y este responde.

Los procesos locales usan la IP 127.0.0.1 (loopback)

Comunicación entre Procesos- Socket

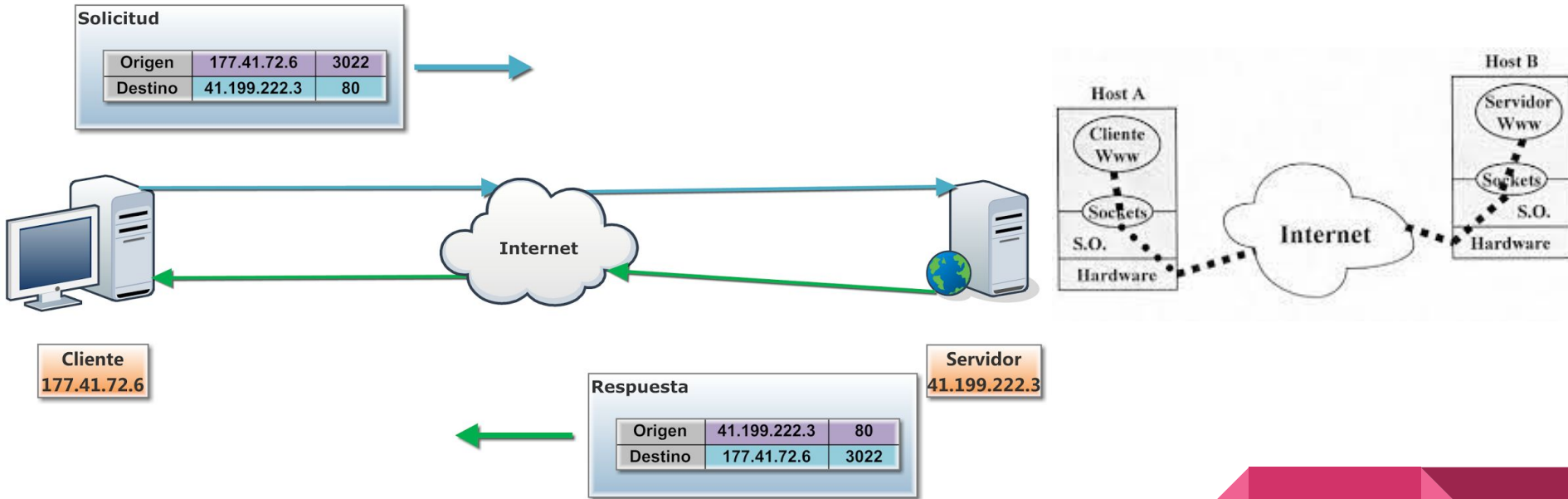


Figura 199: Estructura de aplicaciones Cliente/Servidor en TCP/IP

PROCESO Redirecciones y tuberías

- Todos los procesos para poder lanzarse necesitan tener lo que se conoce como entrada estándar (stdin) y devuelven como resultado dos archivos que son capturados por la terminal en la cual estamos trabajando: la salida estándar (stdout) y el error estándar (stderr).
- Como las salidas stdout y stderr son archivos, al fin y al cabo los podemos trabajar como tales e incluso capturarlos por la terminal. También pueden direccionarse a otros archivos en disco para su posterior inspección.
- Existe además otra forma de trabajar con las salidas, y es transformar la salida stdout de un comando en stdin de otro. Para eso utilizamos el símbolo |, conocido como pipe o tubería.



Carpeta /proc

- Contiene información sobre el sistema, los procesos y el kernel. Se crea cada vez que se inicia el sistema y se actualiza constantemente.
- Contiene Información sobre los procesos en ejecución, Configuración del kernel, Hardware del sistema, Parámetros de configuración.

Cómo está estructurada

- Cada proceso en Linux está representado por un directorio dentro de /proc, nombrado con el ID de proceso (PID)
- Para examinar el directorio en su totalidad se requiere tener privilegios de root
- Algunos de los archivos son propiedad del usuario con el que ejecuta

Ficheros de /proc

- **cwd** es un enlace hacia el directorio de trabajo actual del proceso
- **exe** es un enlace simbólico que contiene el nombre de la ruta actual de la orden ejecutada
- **fd** es un subdirectorío que contiene una entrada por cada fichero que tiene abierto el proceso
- **maps** es un fichero que contiene las regiones de memoria actualmente asociadas y sus permisos de acceso



Tuberías o PIPE

Una tubería (pipe, cauce o '|') consiste en una cadena de procesos conectados de forma tal que la salida de cada elemento de la cadena es la entrada del próximo. Permiten la comunicación y sincronización entre procesos. Es común el uso de buffer de datos entre elementos consecutivos.

Una tubería es unidireccional.

Se utiliza mucho en comando mediante terminales y scripts

```
cat /etc/passwd | grep bash | wc -lines
```

Los comandos “cat”, “grep” y “wc” se lanzan en paralelo y el primero va “alimentando” al segundo, que posteriormente “alimenta” al tercero. Al final tenemos una salida filtrada por esas dos tuberías. Las tuberías empleadas son destruidas al terminar los procesos que las estaban utilizando.

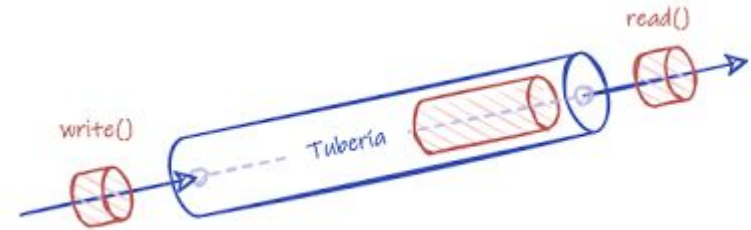


Tuberías o PIPE

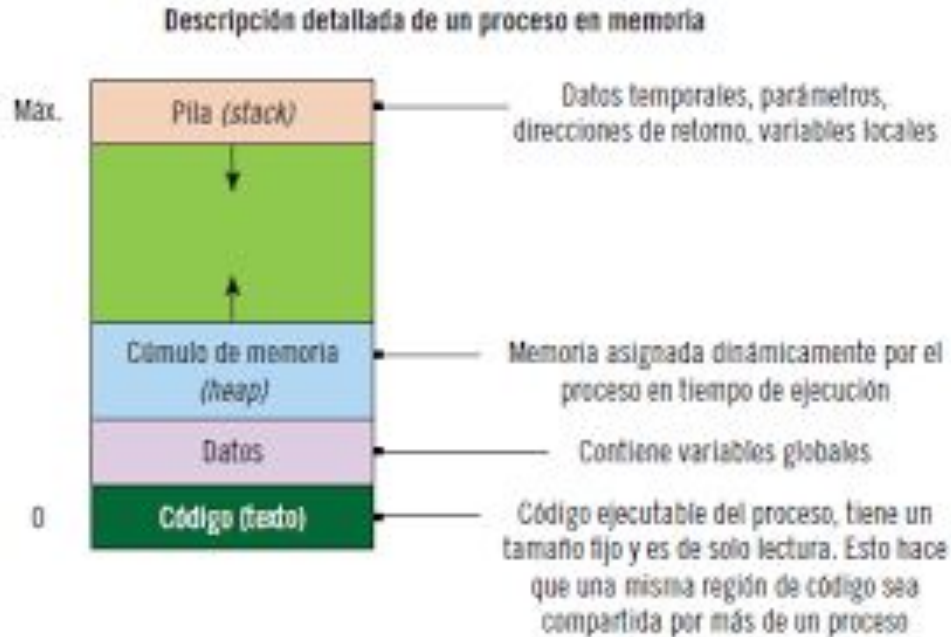
Recordemos, pipes, escritos como "|".

Por ejemplo, qué sucede si escribimos en bash: `echo "la vaca es un poco flaca" | wc -c`

1. Se llama a `echo`, un proceso que escribe su parámetro por stdout.
2. Se llama a `wc -c`, un programa que cuenta cuántos caracteres entran por stdin.
3. Se conecta el stdout de `echo` con el stdin de `wc -c`.



Características de los Procesos



- Un proceso consta de código, datos y pila.
- Los procesos existen en una jerarquía de árbol (varios Hijos, un sólo padre).
- El sistema asigna un identificador de proceso (PID) único al iniciar el proceso.
- El planificador de tareas asigna un tiempo compartido para el proceso según su prioridad (sólo *root* puede cambiar prioridades).

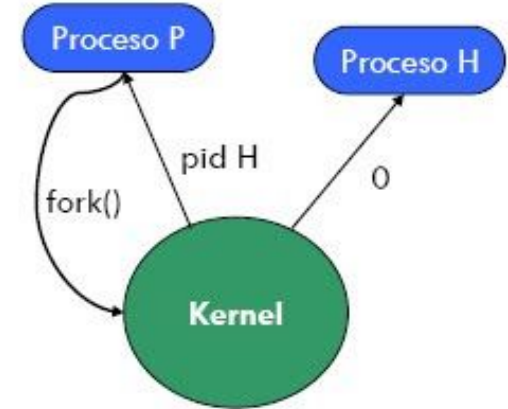
Tipos de Procesos

- ➔ **Ejecución en 1er plano:** proceso iniciado por el usuario o interactivo.
- ➔ **Ejecución en 2o plano:** proceso no interactivo que no necesita ser iniciado por el usuario.
- ➔ **Demonio:** proceso en 2o plano siempre disponible, que da servicio a varias tareas (debe ser propiedad del usuario *root*).
- ➔ **Proceso zombi:** proceso parado que queda en la tabla de procesos hasta que termine su padre. Este hecho se produce cuando el proceso padre no recoge el código de salida del proceso hijo.
- ➔ **Proceso huérfano:** proceso en ejecución cuyo padre ha finalizado. El nuevo identificador de proceso padre (PPID) coincide con el identificador del proceso **init** (1).



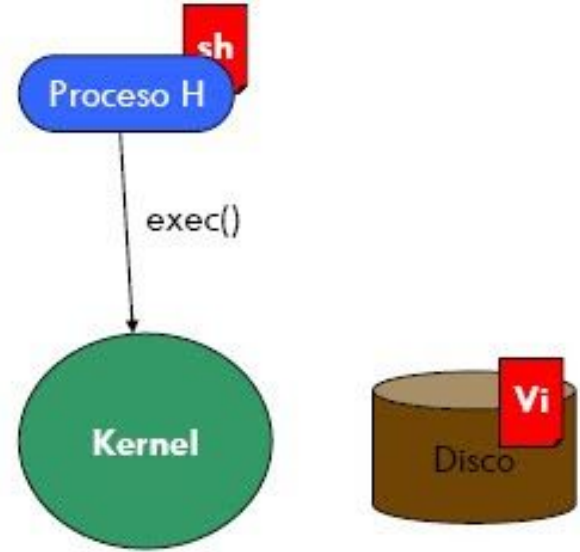
Creación de Procesos en LINUX

- ❑ La llamada al sistema para crear un nuevo proceso se denomina `fork()`.
- ❑ Esta llamada crea una copia casi idéntica del proceso padre.
 - Ambos procesos, padre e hijo, continúan ejecutándose en paralelo.
 - El padre obtiene como resultado de la llamada a `fork()` el pid del hijo y el hijo obtiene 0.
 - Algunos recursos no se heredan (p.ej. señales pendientes).



Creación de Procesos en LINUX

- ❑ El proceso hijo puede invocar la llamada al sistema `exec*()`:
 - sustituye su imagen en memoria por la de un programa diferente
- ❑ El padre puede dedicarse a crear más hijos, o esperar a que termine el hijo:
 - `wait()` lo saca de la cola de “listos” hasta que el hijo termina.



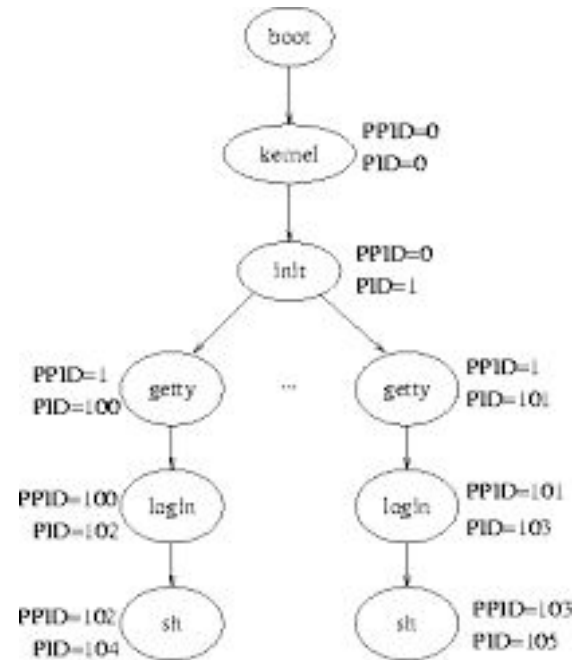
Identificación de procesos Procesos en LINUX

PROCESS ID (PID): Al crearse un nuevo proceso se le asigna un identificador de proceso único. Este número debe utilizarse por el administrador para referirse a un proceso dado al ejecutar un comando.

Los PID son asignados por el sistema a cada nuevo proceso en orden creciente comenzando desde cero. Si antes de un reboot del sistema se llega al nro. máximo, se vuelve a comenzar desde cero, saltando los procesos que aún estén activos.

PARENT PROCESS ID (PPID): La creación de nuevos procesos en Unix se realiza por la vía de duplicar un proceso existente invocando al comando `fork()`. Al proceso original se le llama “padre” y al nuevo proceso “hijo”. El PPID de un proceso es el PID de su proceso padre.

El mecanismo de creación de nuevos procesos en Unix con el comando `fork()` se ve con más detalle en el apartado “Ciclo de vida de un proceso”.



Ciclo de vida de un proceso en Linux

Un proceso se crea invocando a una función del sistema operativo llamada `fork()`. La función `fork()` crea una copia idéntica del proceso que la invoca con excepción de:

- El nuevo proceso tiene un PID diferente
- El PPID del nuevo proceso es el PID del proceso original
- Se reinicia la información de tarificación del proceso (uso de CPU, etc.)

Al retorno de `fork()` se siguen ejecutando las siguientes sentencias del programa en forma concurrente. Para distinguir entre los dos procesos la función `fork()` devuelve un cero al proceso hijo y el PID del nuevo proceso al proceso padre. Normalmente el proceso hijo lanza luego un nuevo programa ejecutando alguna variante de comando `exec()`. En el recuadro puede verse un ejemplo del uso de `fork`.

```
kidpid = fork()
if (kidpid==0) {
    /* soy el hijo, p. ej. lanzo un
    nuevo prog. */
    exec(...)
}
else{
    /* soy el padre */
    ...
    wait()      /* espero exit() del
hijo */
}
```

Ciclo de vida de un proceso en Linux

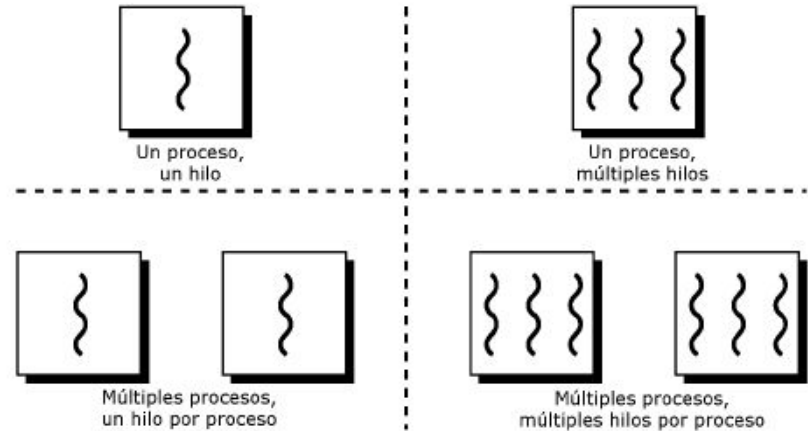
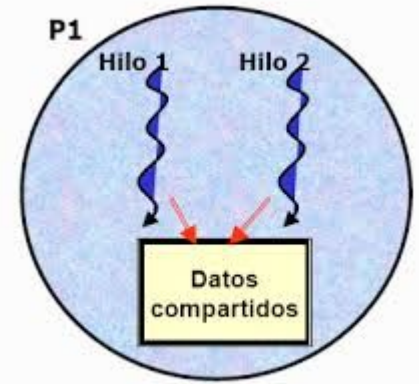
¿quién lanza a correr el primer proceso? Luego del boot del sistema el kernel instala y deja corriendo un proceso llamado **init** con PID=1. Una de las funciones principales de init es lanzar mediante `fork()` intérpretes de comandos que a su vez lanzarán los scripts de inicialización del sistema y los procesos de los usuarios.

Finalizacion de un proceso: Normalmente un proceso termina invocando a la función **exit()** pasando como parámetro un código de salida o **exit code**. El destinatario de ese código de salida es el proceso padre. El proceso padre puede esperar la terminación de su proceso hijo invocando la función **wait()**. Esta función manda al padre a dormir hasta que el hijo ejecute su `exit()` y devuelve el exit code del proceso hijo.

Proceso Zombie: Cuando el proceso hijo termina antes que el padre, el kernel debe conservar el valor del exit code para pasarlo al padre cuando ejecute `wait()`. En esta situación se dice que el proceso hijo está en el estado **zombie**. El kernel devuelve todas las áreas de memoria solicitadas por el proceso pero debe mantener alguna información sobre el proceso (al menos su PID y el exit code). Dado que un proceso zombie no consume recursos fuera de su PID, esto por lo general no provoca problemas.

Procesos ligeros (Hilos o hebras)

- Un hilo de ejecución, es similar a un proceso en que ambos representan una secuencia simple de instrucciones ejecutada en paralelo con otras secuencias.
- Los hilos permiten dividir un programa en dos o más tareas que corren simultáneamente, por medio de la multiprogramación.
- Este método permite incrementar el rendimiento de un procesador de manera considerable.
- En todos los sistemas de hoy en día los hilos son utilizados para simplificar la estructura de un programa que lleva a cabo diferentes funciones.



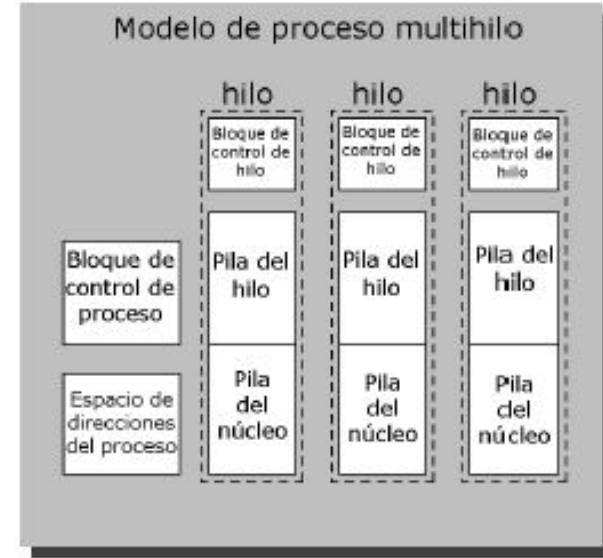
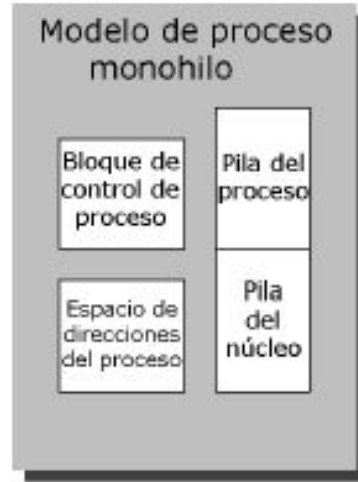
Procesos ligeros (Hilos o hebras)

- Todos los hilos de un proceso comparten los recursos del proceso.
- Residen en el mismo espacio de direcciones y tienen acceso a los mismos datos.
- Cuando un hilo modifica un dato en la memoria, los otros hilos utilizan el resultado cuando acceden al dato.
- Cada hilo tiene su propio estado, su propio contador, su propia pila y su propia copia de los registros de la CPU. Los valores comunes se guardan en el bloque de control de proceso (BCP), y los valores propios en el bloque de control de hilo (TCB).
- Muchos lenguajes de programación (como Java), y otros entornos de desarrollo soportan los llamados hilos o hebras (en inglés, threads).



Procesos ligeros (Hilos o hebras)

- Los hilos son básicamente una tarea que puede ser ejecutada en paralelo con otra tarea; teniendo en cuenta lo que es propio de cada hilo es el contador de programa, la pila de ejecución y el estado de la CPU (incluyendo el valor de los registros).
- En muchos de los sistemas operativos que dan facilidades a los hilos, es más rápido cambiar de un hilo a otro dentro del mismo proceso, que cambiar de un proceso a otro. Este fenómeno se debe a que los hilos comparten datos y espacios de direcciones, mientras que los procesos, al ser independientes, no lo hacen.



Semejanzas entre procesos e hilos

Los hilos operan, en muchos sentidos, igual que los procesos.

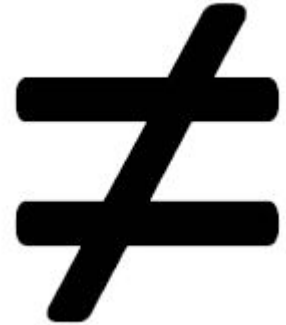
- Pueden estar en uno o varios estados: listo, bloqueado, en ejecución o terminado.
- También comparten la CPU.
- Sólo hay un hilo activo (en ejecución) en un instante dado.
- Un hilo dentro de un proceso se ejecuta secuencialmente.
- Cada hilo tiene su propia pila y contador de programa.
- Pueden crear sus propios hilos hijos.



Diferencias entre procesos e hilos

Los hilos, a diferencia de los procesos, no son independientes entre sí.

- Como todos los hilos pueden acceder a todas las direcciones de la tarea, un hilo puede leer la pila de cualquier otro hilo o escribir sobre ella. Aunque pueda parecer lo contrario la protección no es necesaria ya que el diseño de una tarea con múltiples hilos tiene que ser un usuario único.



Ventaja de usar hilos

- Se tarda mucho menos tiempo en crear un nuevo hilo en un proceso existente que en crear un nuevo proceso.
- Se tarda mucho menos tiempo en terminar un hilo que un proceso.
- Se tarda mucho menos tiempo en conmutar entre hilos de un mismo proceso que entre procesos.
- Los hilos hacen más rápida la comunicación entre procesos, ya que al compartir memoria y recursos, se pueden comunicar entre sí sin invocar el núcleo del SO.



Diferencias Fundamentales entre Hilos y Procesos

Característica	Proceso	Hilo
Espacio de Memoria	Cada proceso tiene su propio espacio de memoria aislado.	Los hilos dentro del mismo proceso comparten el mismo espacio de memoria.
Gestión de Recursos	El sistema operativo asigna recursos (memoria, archivos, etc.) a cada proceso de forma independiente.	Los hilos utilizan los recursos asignados a su proceso padre.
Creación	La creación de un nuevo proceso es una operación pesada que requiere la duplicación de gran parte del contexto del padre.	La creación de un nuevo hilo es una operación ligera con una sobrecarga mínima.
Conmutación de Contexto	La conmutación de contexto entre procesos es costosa ya que implica guardar y restaurar el estado completo de la memoria y los registros.	La conmutación de contexto entre hilos del mismo proceso es más rápida ya que comparten la memoria.



Diferencias Fundamentales entre Hilos y Procesos

Característica	Proceso	Hilo
Comunicación	La comunicación entre procesos (IPC) requiere mecanismos especiales proporcionados por el sistema operativo (por ejemplo, tuberías, memoria compartida, sockets).	La comunicación entre hilos del mismo proceso es sencilla a través de la memoria compartida, aunque requiere sincronización.
Independencia	Los procesos son generalmente independientes; la falla de un proceso no afecta directamente a otros.	Los hilos son dependientes de su proceso padre; si el proceso termina, todos sus hilos también lo hacen.
Sobrecarga	Mayor sobrecarga en términos de recursos y tiempo de gestión.	Menor sobrecarga, lo que permite una mayor eficiencia en la ejecución concurrente.

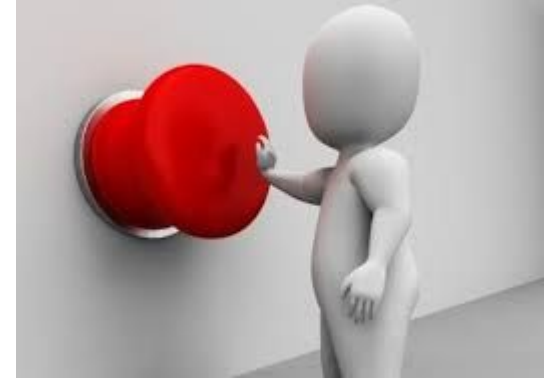


Interrupciones

- Es una señal recibida por el procesador, indicando que debe “interrumpir” el curso de ejecución actual y pasar a ejecutar código específico para tratar la situación.
- Es la suspensión temporal de la ejecución de un proceso, para pasar a ejecutar una subrutina de servicio de interrupción, la cual, por lo general, no forma parte del programa, sino que pertenece al sistema operativo o al BIOS. Una vez finalizada dicha subrutina, se reanuda la ejecución del programa.
- Las interrupciones surgen de la necesidad que tienen los dispositivos periféricos (E/S) de enviar información al procesador principal de un sistema informático.
- La primera técnica que se empleó para esto fue el *polling*, que consistía en que el propio procesador se encargará de sondear los dispositivos periféricos cada cierto tiempo para averiguar si tenía pendiente alguna comunicación para él. Este método presentaba el inconveniente de ser muy ineficiente, ya que el procesador consumía constantemente tiempo y recursos en realizar estas instrucciones de sondeo.
- El mecanismo de interrupciones fue la solución que permitió al procesador desentenderse de esta problemática.

Procesamiento de Interrupciones

1. Terminar la ejecución de la instrucción máquina en curso.
2. Salvar el valor del contador de programa, IP, en la pila, de manera que en la CPU, al terminar el proceso, pueda seguir ejecutando el programa a partir de la última instrucción.
3. La CPU salta a la dirección donde está almacenada la rutina de servicio de interrupción (Interrupt Service Routine, o abreviado ISR) y ejecuta esa rutina que tiene como objetivo atender al dispositivo que generó la interrupción.
4. Una vez que la rutina de la interrupción termina, el procesador restaura el estado que había guardado en la pila en el paso 2 y retorna al programa que se estaba usando anteriormente.



Clases de Interrupciones

- **Interrupciones de hardware:** Pueden producir en cualquier momento independientemente de lo que esté haciendo el CPU en ese momento. Las causas que las producen son externas al procesador y a menudo suelen estar ligadas con los distintos dispositivos de E/S. N
- **Excepciones:** Son aquellas que se producen de forma síncrona a la ejecución del procesador y por tanto podrían predecirse si se analiza con detenimiento la traza del programa que en ese momento estaba siendo ejecutado en la CPU. Normalmente son causadas al realizarse operaciones no permitidas tales como la división entre 0, el desbordamiento, el acceso a una posición de memoria no permitida, etc. Normalmente genera un cambio de contexto a modo supervisor para que el sistema operativo atienda el error.
- **Interrupciones por software:** Las interrupciones por software son aquellas generadas por un programa en ejecución. Para generarlas, existen distintas instrucciones en el código máquina que permiten al programador producir una interrupción, las cuales suelen tener nombres tales como INT (por ejemplo, en DOS se realiza la instrucción INT 0x21 y en Unix se utiliza INT 0x80 para hacer llamadas de sistema).

La Necesidad de Sincronización

Cuando múltiples procesos o hilos acceden a recursos compartidos, como variables en memoria, archivos o dispositivos de E/S, es crucial implementar mecanismos de sincronización para evitar la corrupción de datos y garantizar la correcta ejecución concurrente. Sin una sincronización adecuada, el resultado de la ejecución concurrente puede volverse impredecible y conducir a errores difíciles de depurar. Esto es particularmente relevante para los hilos, ya que comparten el mismo espacio de memoria, lo que facilita el acceso a los datos pero también aumenta el riesgo de interferencia si no se controla adecuadamente. Incluso los procesos, cuando cooperan en una tarea y comparten recursos a través de mecanismos como la memoria compartida, requieren sincronización para coordinar sus acciones.



Condiciones de Competencia

Una condición de competencia, o carrera crítica, ocurre cuando el resultado de un programa depende del orden impredecible en el que múltiples hilos o procesos acceden y manipulan datos compartidos. Un ejemplo clásico es el de dos hilos intentando incrementar un contador compartido simultáneamente. Si ambos hilos leen el valor actual del contador, luego lo incrementan en sus registros locales y finalmente escriben el nuevo valor de vuelta a la memoria, el resultado final puede ser incorrecto.

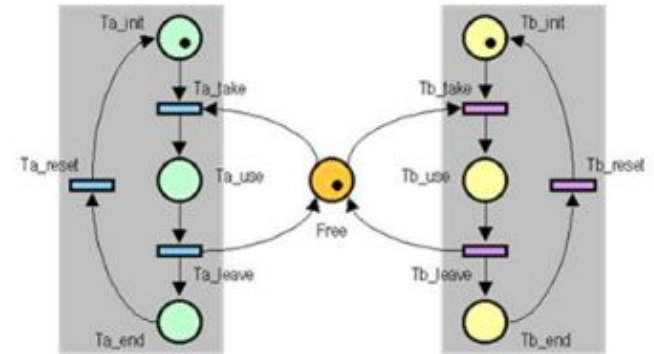


Secciones Críticas y la Importancia de la Exclusión Mutua

Una sección crítica es un segmento de código dentro de un programa donde se accede a recursos compartidos. Para prevenir las condiciones de competencia, es fundamental garantizar la exclusión mutua en el acceso a estas secciones críticas.

La exclusión mutua es un mecanismo que asegura que solo un hilo o proceso pueda ejecutar su sección crítica en un momento dado, impidiendo el acceso concurrente y, por lo tanto, las inconsistencias en los datos.

La identificación y protección de las secciones críticas es fundamental para lograr una ejecución concurrente correcta. La exclusión mutua garantiza la atomicidad de las operaciones dentro de la sección crítica. Al permitir que solo una entidad acceda a la sección crítica a la vez, se asegura que los datos compartidos se manipulen de manera controlada y consistente, evitando así resultados inesperados y errores.



Conceptos

- **Concepto de exclusión mutua:** Consiste en que un solo proceso excluye temporalmente a todos los demás para usar un recurso compartido de forma que garantice la integridad del sistema.
- **Concepto de sección crítica:** Es la parte del programa con un comienzo y un final claramente marcados que generalmente contiene la actualización de una o más variables compartidas.
- **Bloqueo mutuo:** El **bloqueo mutuo** (también conocido como interbloqueo, traba mortal, *deadlock*, abrazo mortal) es el bloqueo permanente de un conjunto de procesos o hilos de ejecución en un sistema concurrente que compiten por recursos del sistema o bien se comunican entre ellos.
- El interbloqueo se produce si cada proceso retiene un recurso y solicita el otro.
- Una posible estrategia para resolver estos interbloqueos es imponer restricciones en el diseño del sistema sobre el orden en el que se solicitan los recursos.

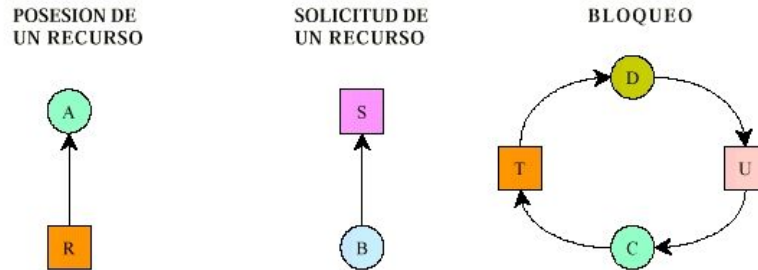


Figura 6.2: Gráficas de asignación de recursos.

Prevención

La estrategia de prevención del interbloqueo consiste, a grandes rasgos, en diseñar un sistema de manera que esté excluida, a priori, la posibilidad de interbloqueo.

Retención y Espera

La condición de retención y espera puede prevenirse exigiendo que todos los procesos soliciten todos los recursos que necesiten a un mismo tiempo y bloqueando el proceso hasta que todos los recursos puedan concederse simultáneamente. Esta solución resulta ineficiente por dos factores.

- Un proceso puede estar suspendido durante mucho tiempo, esperando que se concedan todas sus solicitudes de recursos, cuando de hecho podría haber avanzado con sólo algunos de los recursos.
- Los recursos asignados a un proceso pueden permanecer sin usarse durante periodos considerables, tiempo durante el cual se priva del acceso a otros procesos.



Prevención

No apropiación

La condición de no apropiación puede prevenirse de varias formas.

Primero, si a un proceso que retiene ciertos recursos se le deniega una nueva solicitud, dicho proceso deberá liberar sus recursos anteriores y solicitarlos de nuevo, cuando sea necesario, junto con el recurso adicional.

Por otra parte, si un proceso solicita un recurso que actualmente está retenido por otro proceso, el sistema operativo puede expulsar al segundo proceso y exigirle que libere sus recursos. Este último esquema evitará el interbloqueo sólo si no hay dos procesos que posean la misma prioridad.

Esta técnica es práctica sólo cuando se aplica a recursos cuyo estado puede salvarse y restaurarse más tarde de una forma fácil, como es el caso de un procesador.



Detección

Las estrategias de prevención del interbloqueo son muy conservadoras; solucionan el problema del interbloqueo limitando el acceso a los recursos e imponiendo restricciones a los procesos. En el lado opuesto, las estrategias de detección del interbloqueo no limitan el acceso a los recursos ni restringen las acciones de los procesos. Con detección del interbloqueo, se concederán los recursos que los procesos necesiten siempre que sea posible. Periódicamente, el sistema operativo ejecuta un algoritmo que permite detectar la condición de círculo vicioso de espera descrita en el punto anterior. Puede emplearse cualquier algoritmo de detección de ciclos en grafos dirigidos.

El control del interbloqueo puede llevarse a cabo tan frecuentemente como las solicitudes de recursos o con una frecuencia menor, dependiendo de la probabilidad de que se produzca el interbloqueo.

Una vez detectado el interbloqueo, hace falta alguna estrategia de recuperación. Las técnicas siguientes son posibles enfoques, enumeradas en orden creciente de sofisticación:



Detección

1. Abandonar todos los procesos bloqueados.
2. Retroceder cada proceso interbloqueado hasta algún punto de control definido previamente y volver a ejecutar todos los procesos.
3. Abandonar sucesivamente los procesos bloqueados hasta que deje de haber interbloqueo. El orden en el que se seleccionan los procesos a abandonar seguirá un criterio de mínimo costo. Después de abandonar cada proceso, se debe ejecutar de nuevo el algoritmo de detección para ver si todavía existe interbloqueo.
4. Apropiarse de recursos sucesivamente hasta que deje de haber interbloqueo. Un proceso que pierde un recurso por apropiación debe retroceder hasta un momento anterior a la adquisición de ese recurso.



Recuperación

Un sistema que pretenda recuperarse del interbloqueo, debe invocar a un algoritmo de detección cuando lo considere oportuno (ej. periódicamente)

Formas de intentar la recuperación:

- Terminación de procesos
- matando a todos los procesos implicados (drástico)
- matando a uno de los procesos ¿cuál?
 - el que más recursos libere
 - el que menos tiempo lleve en ejecución...
- Retrocediendo la ejecución de algún proceso (*rollback*) muy complicado de implementar y necesita que el programa esté diseñado para que pueda retroceder



Mecanismos de Sincronización

1. **Semáforos**
2. **Mutexes (Cerrojos)**
3. **Monitores**
4. **Barreras**



Semáforos

Un semáforo es un mecanismo de sincronización que se utiliza generalmente en sistemas con memoria compartida, bien sea un monoprocesador o un multiprocesador. Un semáforo es un objeto con un valor entero al que se le puede asignar un valor inicial no negativo y al que sólo se puede acceder utilizando dos operaciones atómicas: wait y signal (también llamadas down o up, respectivamente).

Las definiciones de estas dos operaciones son las siguientes:

```
wait(s){  
    s = s - 1;  
    if (s < 0)  
        Bloquear al proceso;  
}  
signal(s){  
    s = s + 1;  
    if (s <= 0)  
        Desbloquear a un proceso bloqueado;  
}
```

Semáforo:

Existen 3 estados de los procesos en el CPU



Bloqueado: Deben ocurrir ciertos eventos externos para el proceso pueda ejecutarse.

Listo: Está detenido temporalmente por la ejecución de otro proceso.

En ejecución: Este proceso se está ejecutando en este momento

Semáforos

Se pueden contemplar los semáforos como variables que tienen un valor entero sobre el que se definen las tres operaciones siguientes:

1. Un semáforo puede inicializarse con un valor no negativo.
2. La operación wait decrementa el valor del semáforo. Si el valor se hace negativo, el proceso que ejecuta el wait se bloquea.
3. La operación signal incrementa el valor del semáforo. Si el valor no es positivo, se desbloquea a un proceso bloqueado por una operación wait.

El número de procesos que en un instante determinado se encuentran bloqueados en una operación wait viene dado por el valor absoluto del semáforo si es negativo. Cuando un proceso ejecuta la operación signal, el valor del semáforo se incrementa. En el caso de que haya algún proceso bloqueado en una operación wait anterior, se desbloqueará a un solo proceso.



Mutex

Es un semáforo binario con dos operaciones atómicas:



- **Lock:** intenta bloquear el mutex. Si el mutex ya está bloqueado por otro proceso, el proceso que realiza la operación se bloquea.



- **Unlock:** desbloquea el mutex. Si existen procesos bloqueados en él, se desbloqueará a uno para de ellos que será el nuevo proceso que adquiera el mutex.



Ejemplo Mutex

Fragmento de código

```
Mutex mi_mutex
```

```
Función acceder_recurso_compartido():
```

```
    Adquirir(mi_mutex)  // Bloquear el mutex si está libre; esperar si está bloqueado
```

```
    try:
```

```
        // ... código que accede al recurso compartido ...
```

```
    finally:
```

```
        Liberar(mi_mutex)  // Desbloquear el mutex, permitiendo que otros hilos lo adquieran
```



Monitores - Definición


Un monitor es una construcción de sincronización de alto nivel que encapsula datos compartidos y los procedimientos que operan sobre esos datos. Asegura que solo un hilo pueda estar activo dentro del monitor en un momento dado (exclusión mutua). Los monitores también proporcionan variables de condición para la sincronización. Los monitores ofrecen un enfoque más estructurado y a menudo más seguro para la sincronización en comparación con los semáforos y los mutexes, al manejar implícitamente la exclusión mutua.



Monitores - Analogía

Para comprender mejor el concepto de monitor, podemos recurrir a la analogía de una sala de espera exclusiva, como la de un consultorio médico o un club privado.

El monitor sería como la totalidad de las instalaciones, incluyendo la sala de espera y el despacho del médico (o la sala principal del club). La regla que permite que solo una persona pueda estar dentro del despacho médico (o la sala principal del club) a la vez representa el mutex o lock del monitor. Si alguien está dentro, los demás deben esperar en la sala de espera exterior. Dentro de esta sala de espera principal, podrían existir áreas designadas o reglas específicas para la entrada, como una sección para personas con cita previa y otra para quienes no la tienen. Si el médico sólo atiende a pacientes con cita en ese momento, las personas sin cita deben aguardar en su área designada hasta que la situación cambie, por ejemplo, cuando el médico termine con las citas. Esta analogía ilustra cómo un monitor gestiona el acceso exclusivo a un recurso (el despacho o la sala principal) y cómo permite que los hilos esperen bajo ciertas condiciones antes de poder acceder.



Barrera - definición

Una barrera es una primitiva de sincronización que obliga a un conjunto de procesos o hilos a esperar hasta que todos hayan alcanzado un cierto punto en su ejecución antes de que cualquiera pueda continuar. Las barreras son útiles para sincronizar cálculos paralelos donde todas las partes necesitan completar una fase antes de que pueda comenzar la siguiente. A diferencia de los bloqueos o los semáforos, las barreras no controlan el acceso a los recursos, sino que sincronizan el progreso de múltiples unidades de ejecución.

Funcionamiento y Uso

Cuando un proceso o hilo llega a una barrera, se detiene y espera. Una vez que el número especificado de procesos o hilos ha llegado a la barrera, todos son liberados y pueden continuar su ejecución. A menudo se utiliza un contador para rastrear el número de procesos o hilos que han alcanzado la barrera. Cuando el contador alcanza el número total, todos los procesos en espera reciben una señal para proceder.



Característica	Semáforo	Mutex	Monitor	Barrera
Definición	Variable entera o tipo de dato abstracto utilizado para controlar el acceso a un recurso compartido.	Mecanismo de bloqueo que asegura que solo un proceso/hilo puede acceder a un recurso compartido a la vez.	Construcción de lenguaje que encapsula datos compartidos y procedimientos, proporcionando exclusión mutua y sincronización de condición.	Construcción de sincronización que obliga a un conjunto de procesos/hilos a esperar hasta que todos alcancen un punto específico antes de continuar.
Propósito Principal	Controlar el acceso a un número limitado de recursos; señalización entre procesos; prevenir condiciones de carrera.	Asegurar acceso mutuamente exclusivo a recursos compartidos o secciones críticas de código; prevenir condiciones de carrera.	Regular el acceso a datos compartidos; proporcionar exclusión mutua y sincronización de condición de alto nivel.	Sincronizar la ejecución de un grupo de procesos/hilos, asegurando que todos alcancen un punto específico antes de continuar.
Mecanismo	Operaciones atómicas esperar (P) y señalar (V) que manipulan un contador entero. Puede bloquear procesos en espera.	Operaciones bloquear (adquirir) y desbloquear (liberar). Introduce el concepto de propiedad; solo el propietario puede liberar el bloqueo. Puede bloquear hilos en espera.	Utiliza un mutex para la entrada y variables de condición con operaciones esperar (<code>wait()</code>) y señalar (<code>signal()</code> / <code>notify()</code>) para sincronización más compleja. Los hilos pueden esperar por condiciones.	Los procesos/hilos esperan hasta que un contador interno alcance un valor predefinido (el número total de participantes). Una vez alcanzado, todos los participantes son liberados.
Propiedad	Los semáforos de conteo estándar no tienen propiedad. Los semáforos binarios pueden implicar propiedad por convención.	Tiene propiedad. Solo el hilo que adquirió el mutex puede liberarlo.	Implícita. El hilo que está ejecutando dentro del monitor tiene la propiedad del mutex subyacente.	No tiene el concepto de propiedad; se basa en la participación de un grupo.
Nivel de Abstracción	Bajo. Requiere una gestión cuidadosa por parte del programador para evitar errores como el olvido de liberar o la señalización incorrecta.	Medio. Proporciona una abstracción para la exclusión mutua, pero requiere una gestión explícita de los bloqueos y desbloqueos.	Alto. Encapsula datos y sincronización, a menudo con soporte del lenguaje de programación, lo que facilita la programación concurrente segura.	Alto. Se centra en la coordinación de un grupo de procesos/hilos, abstrayendo los detalles de la gestión individual de bloqueos.
Casos de Uso Comunes	Control de acceso a un pool de recursos, implementación de locks, solución al problema del productor-consumidor, gestión de colas de login, patrones de "pasar el testigo".	Protección de secciones críticas, prevención del acceso simultáneo a memoria compartida, control de acceso a recursos compartidos, serialización de operaciones.	Gestión de recursos compartidos con exclusión mutua y espera basada en condiciones, implementación del problema del búfer limitado, creación de estructuras de datos seguras para hilos.	Sincronización al final de bucles paralelos, coordinación de fases de cómputo en algoritmos paralelos, sincronización en sistemas de paso de mensajes.

Comparativas

Problemas Clásicos de Concurrency



- **El Problema del Productor-Consumidor**
- **El Problema de los Filósofos Comensales**
- **El Problema de los Lectores y Escritores**

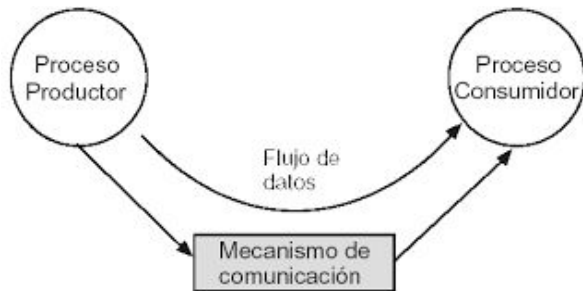
El Problema del Productor-Consumidor

- **Descripción Detallada del Problema**

El problema del productor-consumidor involucra uno o más productores que generan datos y uno o más consumidores que los procesan. Estos comparten un buffer común de tamaño fijo. El productor no debe añadir datos a un buffer lleno, y el consumidor no debe intentar extraer datos de un buffer vacío.

Soluciones Utilizando Semáforos y/o Mutexes

Una solución común para el problema del productor-consumidor utilizando semáforos implica el uso de dos semáforos de conteo: uno para rastrear las ranuras vacías en el buffer (vacío) y otro para rastrear las ranuras llenas (lleno), además de un mutex para proteger el acceso al buffer en sí.



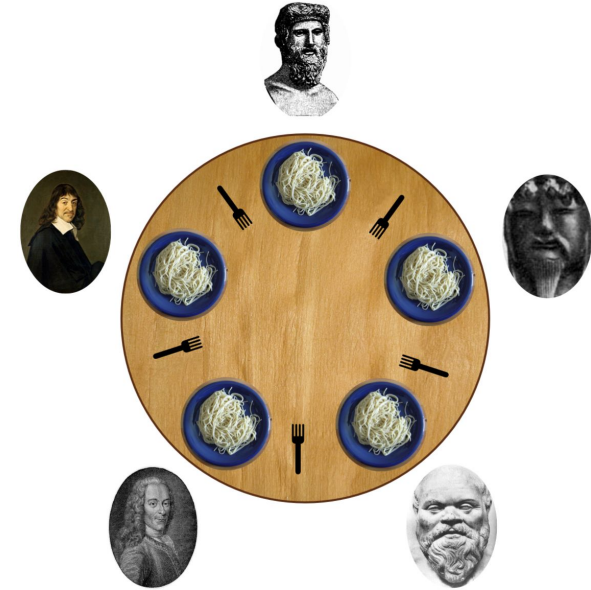
El Problema de los Filósofos Comensales

- **Descripción Detallada del Problema**

Cinco filósofos están sentados alrededor de una mesa redonda comiendo. Entre cada par de filósofos hay un tenedor. Cada filósofo necesita dos tenedores para comer. Alternan entre pensar y comer. El problema consiste en diseñar un protocolo que permita a los filósofos comer sin que se produzca un interbloqueo (todos los filósofos sosteniendo un tenedor y esperando el otro) o inanición (un filósofo nunca llega a comer). Este problema clásico ilustra los desafíos de la asignación de recursos en sistemas concurrentes y el potencial de interbloqueo. Una representación visual de los filósofos y los tenedores alrededor de la mesa es esencial para comprender el problema.

- **Soluciones para Evitar el Interbloqueo**

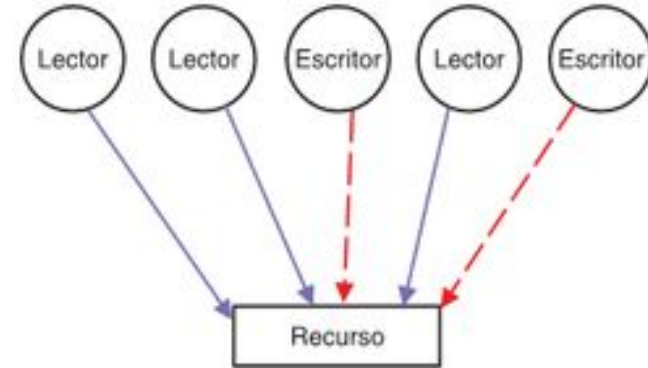
- Limitar el número de filósofos que pueden intentar comer simultáneamente. Si solo se permite que cuatro de los cinco filósofos estén en la fase de intentar tomar los tenedores a la vez, se rompe la condición de espera circular que conduce al interbloqueo.
- Imponer un orden en cómo los filósofos toman los tenedores. Por ejemplo, todos los filósofos podrían intentar tomar primero el tenedor de su izquierda y luego el de su derecha, excepto uno de ellos, que comenzaría por la derecha y luego tomaría el de la izquierda. Este enfoque rompe la simetría que permite que ocurra el interbloqueo.



El Problema de los Lectores y Escritores

- **Descripción Detallada del Problema**

El problema de los lectores y escritores involucra múltiples procesos que desean acceder a un recurso compartido, como una base de datos. Algunos procesos son lectores (solo leen los datos) y otros son escritores (modifican los datos). Se permite que múltiples lectores accedan al recurso simultáneamente, pero solo un escritor puede acceder a él a la vez, y ningún lector puede estar presente mientras un escritor está escribiendo. Este problema destaca la necesidad de equilibrar la concurrencia (permitir múltiples lectores) con la integridad de los datos (acceso exclusivo para los escritores). Una representación visual mostrando lectores y escritores accediendo a un recurso compartido sería útil.



- **Soluciones Utilizando Semáforos y/o Mutexes**

Una solución para el problema de los lectores y escritores que da prioridad a los lectores se puede implementar utilizando un mutex para proteger una variable que cuenta el número de lectores activos y un semáforo para controlar el acceso de los escritores.

Niveles, objetivos y criterios de planificación.

Se consideran tres niveles importantes de planificación, los que se detallan a continuación:

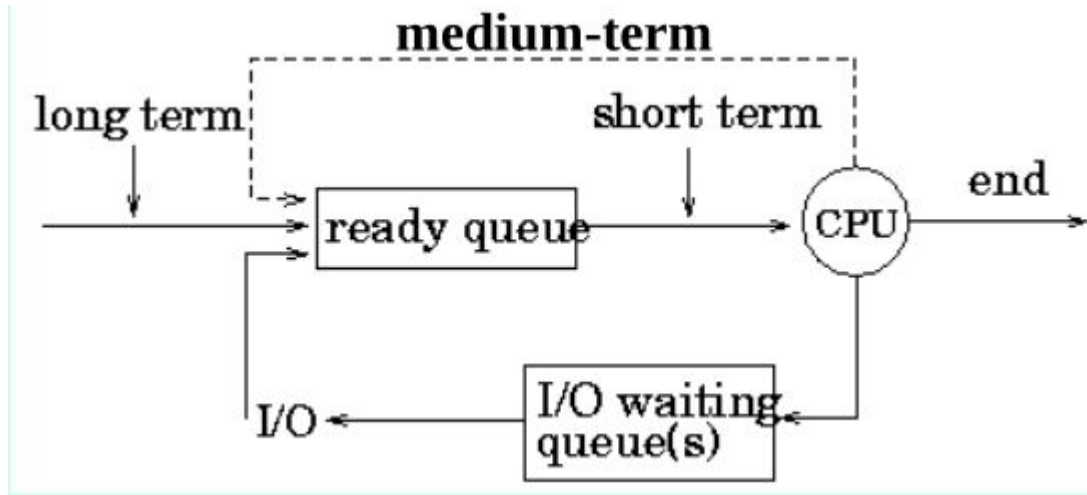
- **Planificación de alto nivel:** Se encarga de llevar procesos Bloqueados o suspendidos a listo. Mantiene las colas de procesos bloqueados y suspendidos.
- **Planificación de nivel intermedio:** Encuentra ventajoso retirar trabajos activos de la memoria para reducir el grado de multiprogramación, y por lo tanto, permitir que los trabajos se completen más aprisa. Determina a qué procesos se les puede permitir competir por la cpu. Efectúa “suspensiones” y “activaciones” de procesos. Nivelar la carga del sistema (procesos activos y pasivos).
- **Planificación de bajo nivel:** Se encarga de pasar de un proceso a otro en memoria principal. Determinando a cuál proceso listo se le asignará el CPU cuando éste se encuentra disponible y asigna la cpu al mismo, es decir que “despacha” la cpu al proceso.

:




Niveles, objetivos y criterios de planificación.

- **Long term:** O Job Scheduler. Selecciona de la cola de trabajos cual pasará a la cola de listos.
- **Short term:** O CPU Scheduler. Selecciona el próximo proceso a ejecutarse y le asigna la CPU.
- **Medium term:** Realiza swap in - swap out entre el disco y la memoria.



Objetivos de planificación.

- **Ser justa:** Todos los procesos son tratados de igual manera.
 - **Maximizar la capacidad de ejecución:** Maximizar el número de procesos servidos por unidad de tiempo.
 - Maximizar el número de usuarios interactivos que reciban unos tiempos de respuesta aceptables.
 - **Ser predecible**
 - **Minimizar la sobrecarga:** No suele considerarse un objetivo muy importante.
 - **Equilibrar el uso de recursos:** Favorecer a los procesos que utilizarán recursos infrautilizados.
 - **Equilibrar respuesta y utilización**
 - **Evitar la postergación indefinida**
 - **Asegurar la prioridad:** Los mecanismos de planificación deben favorecer a los procesos con prioridades más altas.
 - Dar preferencia a los procesos que mantienen recursos claves
- 

Criterios de Planificación

- **Equidad:** Garantizar que cada proceso obtiene su proporción justa de la cpu.
- **Eficacia:** Mantener ocupada la cpu el ciento por ciento del tiempo.
- **Tiempo de respuesta:** Minimizar el tiempo de respuesta para los usuarios interactivos.
- **Tiempo de regreso:** Minimizar el tiempo que deben esperar los usuarios por lotes (batch) para obtener sus resultados.
- **Rendimiento:** Maximizar el número de tareas procesadas por hora.



Técnicas de administración del planificador.



CRITERIOS PARA ALGORITMOS DE PLANIFICACIÓN

- **Utilización de CPU:** en porcentaje.
- **Throughput:** Productividad, cantidad de procesos completados por unidad de tiempo.
- **Turnaround time:** Tiempo de retorno. Desde que el proceso ingresa al sistema hasta que sale.
- **Waiting time:** Tiempo de espera. En la cola de listos.
- **Response time:** Tiempo de respuesta. En un sistema interactivo. El tiempo transcurrido hasta que comienza la respuesta.



FIFO First Input First Output o FCFS First Come First Served

Cuando se tiene que elegir a qué proceso asignar la CPU se escoge al que llevará más tiempo listo. El proceso se mantiene en la CPU hasta que se bloquea voluntariamente.

La ventaja de este algoritmo es su fácil implementación, sin embargo, no es válido para entornos interactivos ya que un proceso de mucho cálculo de CPU hace aumentar el tiempo de espera de los demás procesos . Para implementar el algoritmo sólo se necesita mantener una cola con los procesos listos ordenada por tiempo de llegada. Cuando un proceso pasa de bloqueado a listo se sitúa el último de la cola.



FIFO o FCFS

En esta política de planificación, el procesador ejecuta cada proceso hasta que termina, por tanto, los procesos que en cola de procesos preparados permanecerán encolados en el orden en que lleguen hasta que les toque su ejecución. Este método se conoce también como FIFO (first input, first output, Primero en llegar primero en salir).

Sus características son:

- No apropiativa.
- Es justa, aunque los procesos largos hacen esperar mucho a los cortos.
- Predecible.
- El tiempo medio de servicio es muy variable en función del número de procesos y su duración.

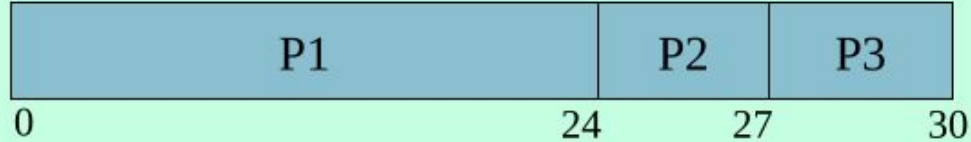


FIFO o FCFS

Proceso Ráfaga de CPU

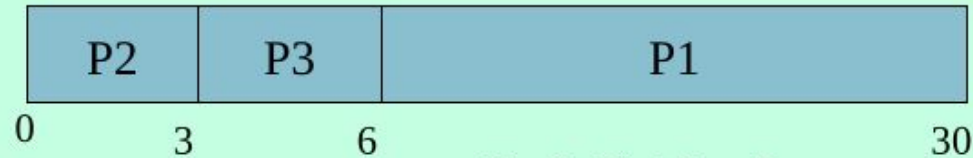
P1	24
P2	3
P3	3

Los procesos llegan en el orden
P1, P2, P3 El diagrama de Gantt
es el siguiente:



$(0+24+27) / 3 = 17$ tiempo de espera promedio

Si cambia el orden de llegada:



$(6+0+3) / 3 = 3$

SJF (Primero el trabajo más corto)

- En este algoritmo, da bastante prioridad a los procesos más cortos a la hora de ejecución y los coloca en la cola.

Ejemplo: Una cola de personas en un supermercado delante de la caja, la persona que menos compra lleva esa pasa primero.

- Este algoritmo selecciona al proceso con el próximo tiempo ejecución más corto. El proceso corto saltará a la cabeza de la cola. El problema está en conocer dichos valores, pero podemos predecirlos usando la información de los ciclos anteriores ejecutados
- La ventaja que presenta este algoritmo sobre el algoritmo FIFO es que minimiza el tiempo de finalización promedio



SJF (Primero el trabajo más corto)

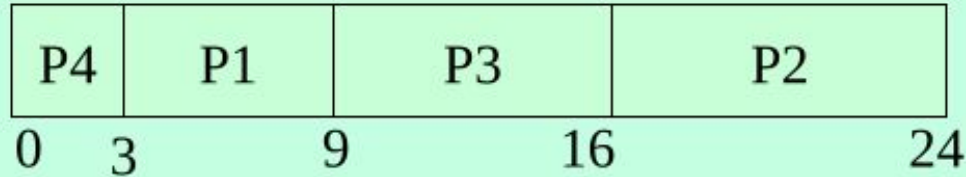
- Asocia a cada proceso la longitud de la próxima ráfaga de CPU.
- Usa esta longitud para planificar al trabajo de más corta duración.
- Hay dos esquemas:
 - **No apropiativo:** Una vez que un proceso tiene la CPU no puede ser desalojado.
 - **Apropiativo:** Si llega un proceso cuya ráfaga es menor que lo que le falta al proceso que está corriendo, lo desaloja de la CPU. Este se conoce como Primero el de menor resto. Shortest - Remaining Time First.



SJF (Primero el trabajo más corto)

Proceso Ráfaga de CPU

P1	6
P2	8
P3	7
P4	3



$$(3 + 16 + 9 + 0) / 4 = 7 \text{ milisegundos}$$

Con FCFS sería $(0 + 6 + 14 + 21) / 4 = 10.25$

SJF No Apropiativo

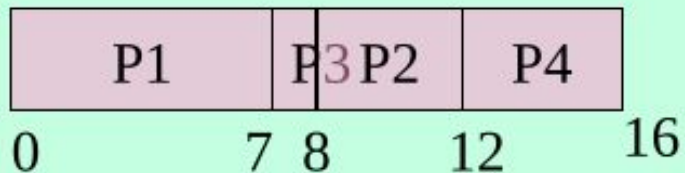
Proceso Arribo R. CPU

P1 0 7

P2 2 4

P3 4 1

P4 5 4



Tiempo promedio de espera = $(0 + 6 + 3 + 7) / 4 = 4$

SRTF “Short Remaining Time First”

Es similar al SJF pero no apropiativo, con la diferencia de que si un nuevo proceso pasa a listo se activa el dispatcher para ver si es más corto que lo que queda por ejecutar del proceso en ejecución. Si es así, el proceso en ejecución pasa a listo y su tiempo de estimación se decrementa con el tiempo que ha estado ejecutándose.

Desventajas:

- El intervalo de CPU es difícil de predecir
- Posibilidad de inanición: los trabajos largos no se ejecutarán mientras haya trabajos cortos.



SJF Apropiativo

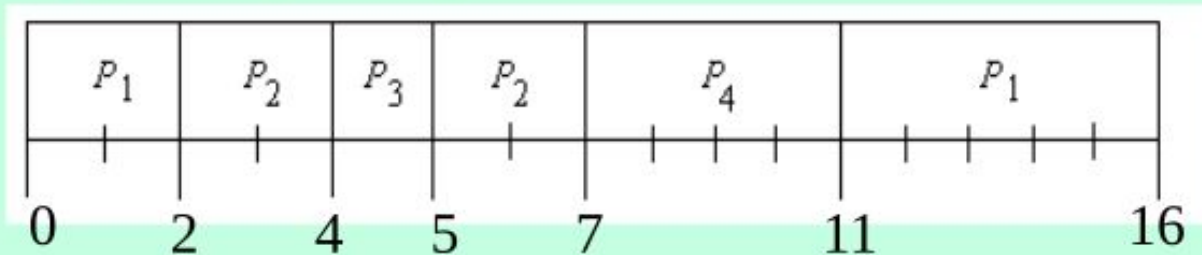
Proceso Arribo R. CPU

P1 0 7

P2 2 4

P3 4 1

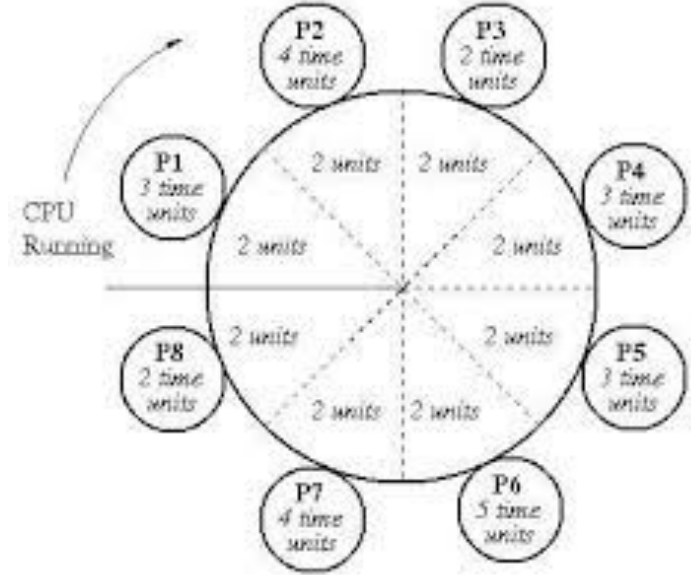
P4 5 4



$$\text{Tiempo promedio de espera} = (9 + 1 + 0 + 2) / 4 = 3$$

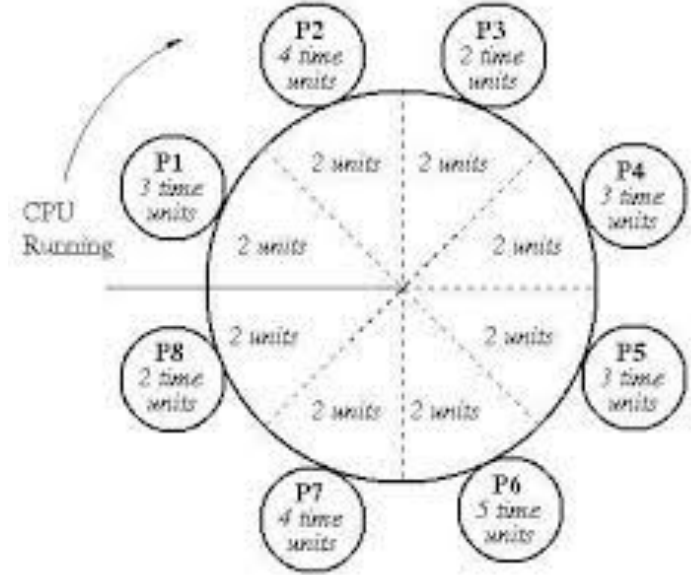
Round Robin

- Este es uno de los algoritmos más antiguos, sencillos y equitativos en el reparto de la CPU entre los procesos.
- Cada proceso tiene asignado un intervalo de tiempo de ejecución, llamado quantum o cuanto. Si el proceso agota su quantum de tiempo, se elige a otro proceso para ocupar la CPU. Si el proceso se bloquea o termina antes de agotar su quantum también se alterna el uso de la CPU. El round robin es muy fácil de implementar. Todo lo que necesita el planificador es mantener una lista de los procesos listos.
- La cola de procesos se estructura como una cola circular. El planificador la recorre asignando un cuanto de tiempo a cada proceso. La organización de la cola es FIFO.



Round Robin

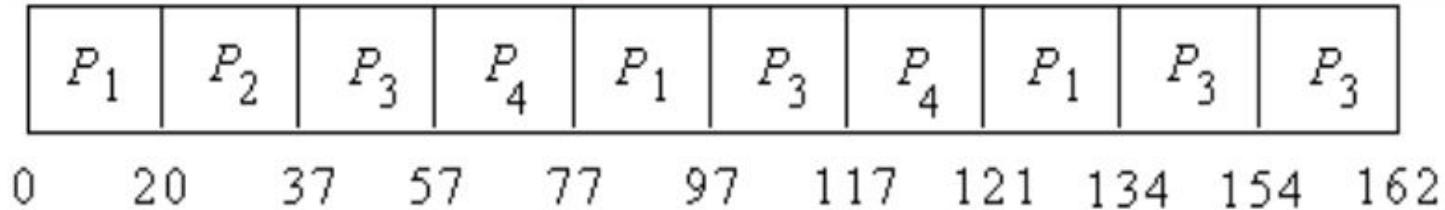
- Cada proceso tiene una pequeña unidad de tiempo de CPU llamada Quantum (q) . Usualmente entre 10 y 100 milisegundos. Después de transcurrido ese tiempo el proceso es desalojado de la CPU y colocado en la cola de listos.
- Si hay n procesos en la cola de listos y el quantum es q , entonces cada procesos toma $1/n$ del tiempo de CPU en porciones de tiempo q unidades a la vez. Ningún proceso espera más de $(n-1)q$ unidades de tiempo.
- Performance
 - si q es grande \Rightarrow FCFS
 - si q es pequeño \Rightarrow q debe ser bastante mayor que el tiempo de cambio de contexto, de otra forma el overhead será grande.



Round Robin

Proceso Ráfaga de CPU

P1	53
P2	17
P3	68
P4	24



- Típicamente, mayor tiempo promedio de retorno (turnaround) pero mejor tiempo de respuesta

Planificación por prioridades

La prioridad es un entero asociado a cada proceso.

La CPU se asigna al proceso de mayor prioridad.

- apropiativo
- no apropiativo

El SJF es un algoritmo por prioridades donde la prioridad la da el próximo tiempo de ráfaga de CPU.

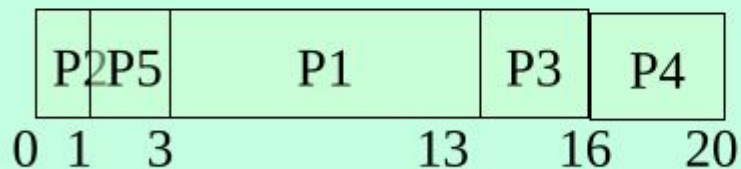
Problema: bloqueo indefinido o inanición.

Solución: envejecimiento.



Planificación por prioridades

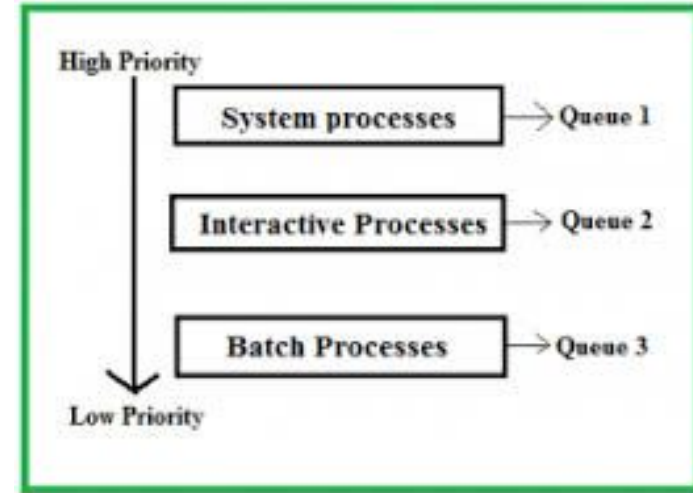
Proceso	Prioridad	R. CPU
P1	3	10
P2	1	1
P3	3	3
P4	4	4
P5	2	2



$$\text{Tiempo medio de espera} = (3 + 0 + 13 + 16 + 1) / 5 = 6.6$$

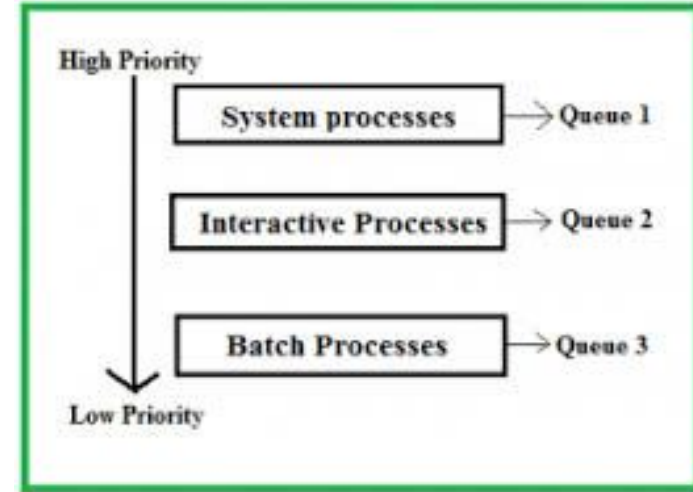
Queues multi-level

- Divide la cola de procesos preparados en varias colas distintas. Los procesos se asignan permanentemente a una cola.
- Por ejemplo, pueden emplearse colas distintas para los procesos de primer plano y de segundo plano.
- Los procesos son asignados permanentemente a una de las colas.
- Cada cola tendrá su propio algoritmo de planificación.
- Además, se debe tener una estrategia de planificación entre las diferentes colas. Por ejemplo, una cola tendrá prioridad sobre otra.



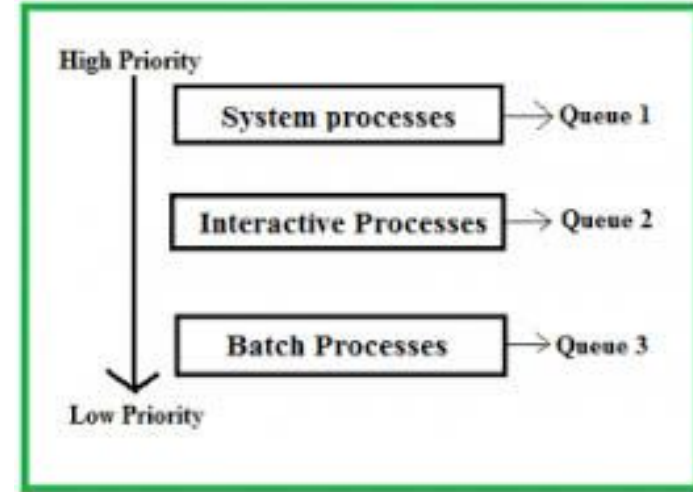
Queues multi-level

- La cola de listos es particionada en colas separadas
 - foreground (frente)
 - background (fondo)
- Cada cola tiene su propio algoritmo de planificación
 - foreground RR
 - background FCFS
- Debe haber una planificación entre colas
- Se fija una prioridad de planificación. Por ejemplo: Se atiende todos los procesos del foreground y luego los del background. Existe la posibilidad de inanición.
- Se fija una porción de tiempo para cada cola. Y durante el mismo cada cola planifica su proceso.
- 80% para foreground
- 20% para background



Queues multi-level

- Un proceso puede moverse entre distintas colas. El envejecimiento puede implementarse de esta forma.
- Para este algoritmo deben definirse los siguientes parámetros:
 - Número de colas
 - Algoritmo de cada cola
 - Método para determinar cuando un proceso asciende
 - Método usado para determinar cuando un proceso desciende
 - Método usado para determinar a qué cola entra un proceso cuando necesita servicio



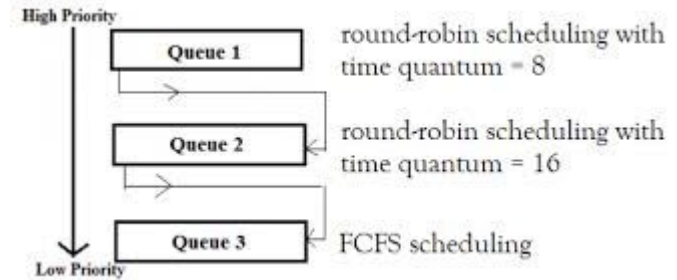
Múltiples colas con retroalimentación

Se diferencia con el anterior en que procesos pueden cambiar de cola (nivel). Se basa en categorizar los procesos según el uso de CPU que tengan. Se garantiza que los procesos con poco uso de procesador tengan mayor prioridad, y los que consumen mucho procesador tendrán baja prioridad.

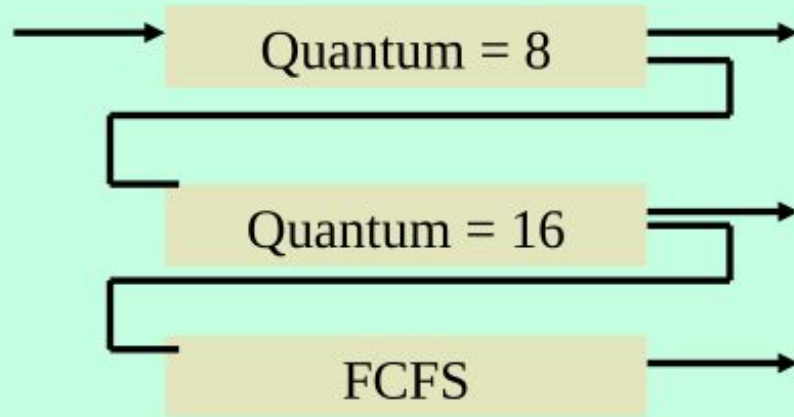
Los procesos, según el consumo de CPU que hagan, serán promovidos a una cola de mayor prioridad o rebajados a una de menor prioridad.

Un planificador Multilevel-Feedback-Queue es definido por:

- El número de colas.
- El algoritmo de planificación para cada cola.
- El método utilizado para promover a un proceso a una cola de mayor prioridad.
- El método utilizado para bajar a un proceso a una cola de menor prioridad.
- El método utilizado para determinar a qué cola será asignado un proceso cuando este *pronto*.



Múltiples colas con retroalimentación - EJEMPLO



- Planificación
 - Todos los jobs entran a la cola 0 y son atendidos RR. Tiene 8 milisegundos, si no terminan pasan a la de abajo.
 - En la segunda cola reciben 16 milisegundos si no terminan pasan a la de abajo.
 - ¿Cuál es la prioridad que asigna este algoritmo?


RUNLEVELS

El **runlevel** (del inglés, **nivel de ejecución**) es cada uno de los estados de ejecución en que se puede encontrar el sistema Linux. Existen 7 niveles de ejecución en total:

- **Nivel de ejecución 0:** Apagado.
- **Nivel de ejecución 1:** Monousuario (sólo usuario root; no es necesaria la contraseña). Se suele usar para analizar y reparar problemas.
- **Nivel de ejecución 2:** Multiusuario sin soporte de red.
- **Nivel de ejecución 3:** Multiusuario con soporte de red.
- **Nivel de ejecución 4:** *Como el runlevel 3, pero no se suele usar*
- **Nivel de ejecución 5:** Multiusuario en modo gráfico (X Windows).
- **Nivel de ejecución 6:** Reinicio.

Runlevel	Systemd Description
0	poweroff.target
1	rescue.target
2	multi-user.target
3	multi-user.target
4	multi-user.target
5	graphical.target
6	reboot.target

Este sistema de niveles de ejecución lo proporciona el sistema de arranque por defecto de la mayoría de distribuciones GNU/Linux (**init**).



Cambiar de RUNLEVEL

Existe una utilidad para línea de comandos que permite cambiar de un nivel de ejecución a otro. Esta es la herramienta *init*. Para cambiar de nivel de ejecución sólo hay que ejecutar *init* seguido del número del runlevel.

Por ejemplo:

- ***init 0***: Cambia al runlevel 0 (se apaga el sistema, equivalente al comando *halt*).
- ***init 2***: Cambia al runlevel 2.
- ***init 6***: Cambia al runlevel 6 (reinicia el sistema, equivalente al comando *reboot*).

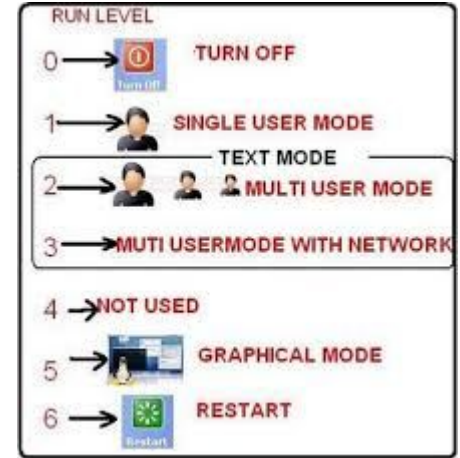


Modificar RUNLEVEL por defecto

- Por defecto, el sistema suele arrancar en el nivel de ejecución 5 (modo gráfico). Si se quisiera modificar este comportamiento, habría que editar el fichero ***/etc/inittab*** la línea:

id:5:initdefault:

donde el número 5 indica que el nivel de ejecución por defecto es el 5. Este número es el que hay que modificar para cambiar el nivel de ejecución en el que arranca el sistema por defecto.



COMANDOS en LINUX



Comando Procesos - pstree

- Si la utilizamos con la opción -p, podremos ver el Process Identifier (o identificador de proceso, comúnmente llamado PID) de cada proceso. En este contexto, decimos que si el proceso A es el padre de B, el PID de A recibe el nombre de PPID (Parent Process ID) de B.
- Si a continuación de la opción -p indicamos un PID dado, el árbol de procesos se mostrará comenzando por el proceso al que le corresponde dicho PID.

```
curso:~# pstree -p
systemd(1)─cron(370)
           └─dbus-daemon(353)
           └─dhclient(400)
           └─dhclient(419)
           └─login(456)─bash(471)─pstree(877)
           └─rsyslogd(352)─{in:imklog}(368)
                        └─{in:imuxsock}(367)
                        └─{rs:main Q:Reg}(369)
           └─sshd(458)
           └─systemd(468)─(sd-pam)(469)
           └─systemd-journal(163)
           └─systemd-logind(351)
           └─systemd-timesyn(398)─{sd-resolve}(399)
           └─systemd-udev(195)
```

curso:~#

```
curso:~# pstree -p 352
rsyslogd(352)─{in:imklog}(368)
             └─{in:imuxsock}(367)
             └─{rs:main Q:Reg}(369)
```

curso:~# _

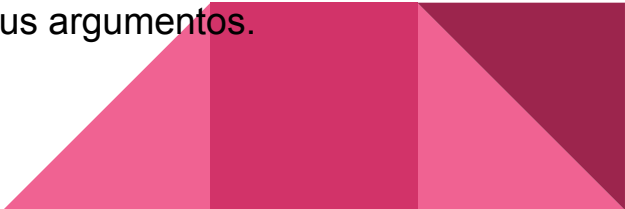
Comando Procesos - ps

- El uso de las opciones **-ef** o **aux** (esta última sin guión medio al comienzo) hace que podamos visualizar rápidamente el listado de procesos que están corriendo en nuestro sistema.
- Una característica distintiva de **ps** es que no es interactivo, sino que saca una foto de los procesos que están corriendo en un determinado momento.
- Sin opciones, **ps** nos devuelve los procesos ligados a la terminal actual.
- El uso de las opciones a, u, y x de manera separada devuelve una lista de:
 - a) todos los procesos que se están ejecutando en una terminal,
 - u) muestra el estado de los procesos del usuario actual y qué cantidad de recursos está requiriendo cada uno, y
 - x) indica la información de los demonios y procesos sin terminal. Al combinar estas opciones como aux podemos acceder a todos esos datos simultáneamente.

```
gmart@informatica:~$ ps -aux
```

USER	PID	%CPU	%MEM	VSZ	RSS	TTY	STAT	START	TIME	COMMAND
root	1	0.0	0.0	23024	13524	?	Ss	08:04	0:07	/sbin/init sp
root	2	0.0	0.0	0	0	?	S	08:04	0:00	[kthreadd]
root	3	0.0	0.0	0	0	?	S	08:04	0:00	[pool_workque
root	4	0.0	0.0	0	0	?	I<	08:04	0:00	[kworker/R-rc
root	5	0.0	0.0	0	0	?	I<	08:04	0:00	[kworker/R-rc
root	6	0.0	0.0	0	0	?	I<	08:04	0:00	[kworker/R-sl
root	7	0.0	0.0	0	0	?	I<	08:04	0:00	[kworker/R-ne
root	10	0.0	0.0	0	0	?	I<	08:04	0:00	[kworker/0:0H
root	12	0.0	0.0	0	0	?	I<	08:04	0:00	[kworker/R-mm
root	13	0.0	0.0	0	0	?	I	08:04	0:00	[rcu_tasks_kt

Comando Procesos - ps (continuación)

- **USER:** usuario dueño del proceso.
 - **PID:** id del proceso
 - **%CPU:** porcentaje de tiempo de CPU utilizado sobre el tiempo que el proceso ha estado en ejecución.
 - **%MEM:** porcentaje de memoria física utilizada.
 - **VSZ:** memoria virtual del proceso medida en KiB.
 - **RSS (Resident Set Size):** es la cantidad de memoria física no swapeada que la tarea ha utilizado (en KiB)
 - **TT:** terminal asociada al proceso. Si en este campo vemos un signo de pregunta (?) significa que el proceso en cuestión se trata de un servicio que no está utilizando ninguna terminal.
 - **STAT:** código de estado. Una R en este campo indica que el proceso se está ejecutando. Una S nos dice que está durmiendo esperando a que suceda un evento.
 - **STARTED:** fecha y hora de inicio del proceso.
 - **TIME:** tiempo de CPU acumulado.
 - **COMMAND:** comando relacionado con el proceso, incluyendo todos sus argumentos.
- 

Comando Procesos - ps (continuación)

- Es importante notar que ps nos permite adaptar la cantidad y el orden de las columnas a mostrar, e inclusive nos deja ordenar el resultado utilizando una de ellas de manera ascendente o descendente. Eso es posible mediante las opciones -eo y --sort, respectivamente. La primera debe ser seguida por la lista de campos a mostrar (tomados de la sección STANDARD FORMAT SPECIFIERS del man page)
- La segunda debemos colocar un signo igual y el criterio de ordenamiento (-%mem indica ordenar por uso de memoria en forma descendente):

ps -eo pid,ppid,cmd,%cpu,%mem --sort=-%mem

PID	PPID	CMD	%CPU	%MEM
1	0	/sbin/init	0.0	0.6
458	1	/usr/sbin/sshd -D	0.0	0.5
468	1	/lib/systemd/systemd --user	0.0	0.5
471	456	-bash	0.0	0.4
351	1	/lib/systemd/systemd-logind	0.0	0.4
163	1	/lib/systemd/systemd-journald	0.0	0.4
195	1	/lib/systemd/systemd-udev	0.0	0.3
398	1	/lib/systemd/systemd-timesyncd	0.0	0.3
353	1	/usr/bin/dbus-daemon --system	0.0	0.3
456	1	/bin/login --	0.0	0.3
917	471	ps -eo pid,ppid,cmd,%cpu,%mem	0.0	0.3

Comando Procesos - kill y killall

- Matar un proceso significa interrumpir la normal ejecución del mismo.
- Para evitar errores, antes de matar procesos es preferible identificar aquellos que se verían afectados por nuestra medida. Por ejemplo, antes de matar el proceso con PID 3092 es una buena idea identificarlo primero mediante ps y la opción **--pid**.
- Para identificar y matar varios procesos de una sola vez:
- Identifiquemos aquellos procesos que comparten el padre cuyo PID es 1576: **pgrep -l -P 1576**
- Antes de reemplazar pgrep por pkill, preguntémonos si realmente deseamos matar todos esos procesos. Si la respuesta es sí, podemos hacer **pkill -P 1576**
- Si en vez de realizar la identificación por PPID quisiéramos hacerlo a partir del propietario o del grupo dueño del proceso, simplemente deberemos reemplazar la opción -P con -u o -G, respectivamente, seguido del nombre o del identificador del usuario o grupo.
- En el caso de que existan varios procesos que compartan el mismo nombre y deseemos detenerlos a todos, podemos hacer uso del comando killall



Señales

- Cuando empleamos **kill** para matar un proceso, se le envía al mismo una señal. En otras palabras, le estamos mandando una indicación para modificar su funcionamiento normal. Los distintos tipos de señales se enumeran a través de **kill -l**
- Las señales más utilizadas son **1 (SIGHUP)**, **9 (SIGKILL)**, y **15 (SIGTERM)**:
 - Cuando se cierra la terminal asociada a uno o más procesos, se envía a los mismos la señal **SIGHUP**, lo que hace que se detengan.
- **SIGKILL** le indica a un proceso que debe finalizar de inmediato, sin darle tiempo a liberar adecuadamente los recursos que esté utilizando. Esta señal no puede ser ignorada por el proceso al que es enviada.
- **SIGTERM** le permite al proceso terminar su ejecución normalmente dándole la oportunidad de liberar los recursos utilizados. Aunque esto suene bien, cabe aclarar que esta señal puede ser ignorada por un proceso. Por eso puede ser necesario (como último recurso) utilizar SIGKILL en algunas ocasiones.



Señales

```
curso:~# kill -l
```

1) SIGHUP	2) SIGINT	3) SIGQUIT	4) SIGILL	5) SIGTRAP
6) SIGABRT	7) SIGBUS	8) SIGFPE	9) SIGKILL	10) SIGUSR1
11) SIGSEGV	12) SIGUSR2	13) SIGPIPE	14) SIGALRM	15) SIGTERM
16) SIGSTKFLT	17) SIGCHLD	18) SIGCONT	19) SIGSTOP	20) SIGTSTP
21) SIGTTIN	22) SIGTTOU	23) SIGURG	24) SIGXCPU	25) SIGXFSZ
26) SIGVTALRM	27) SIGPROF	28) SIGWINCH	29) SIGIO	30) SIGPWR
31) SIGSYS	34) SIGRTMIN	35) SIGRTMIN+1	36) SIGRTMIN+2	37) SIGRTMIN+3
38) SIGRTMIN+4	39) SIGRTMIN+5	40) SIGRTMIN+6	41) SIGRTMIN+7	42) SIGRTMIN+8
43) SIGRTMIN+9	44) SIGRTMIN+10	45) SIGRTMIN+11	46) SIGRTMIN+12	47) SIGRTMIN+13
48) SIGRTMIN+14	49) SIGRTMIN+15	50) SIGRTMAX-14	51) SIGRTMAX-13	52) SIGRTMAX-12
53) SIGRTMAX-11	54) SIGRTMAX-10	55) SIGRTMAX-9	56) SIGRTMAX-8	57) SIGRTMAX-7
58) SIGRTMAX-6	59) SIGRTMAX-5	60) SIGRTMAX-4	61) SIGRTMAX-3	62) SIGRTMAX-2
63) SIGRTMAX-1	64) SIGRTMAX			

```
curso:~# _
```

Comando top

- A diferencia de ps, top actualiza la información cada cierto tiempo (por defecto, cada tres segundos) y además provee otros datos útiles sobre el estado del sistema.

```
top - 11:09:14 up 37 min,  1 user,  load average: 0,00, 0,00, 0,00
Tasks:  62 total,   1 running,  61 sleeping,   0 stopped,   0 zombie
%Cpu(s):  0,3 us,  0,0 sy,  0,0 ni, 99,7 id,  0,0 wa,  0,0 hi,  0,0 si,  0,0 st
KiB Mem : 1020416 total,  712696 free,   48356 used,  259364 buff/cache
KiB Swap: 1046524 total, 1046524 free,    0 used.  829516 avail Mem
```

PID	USER	PR	NI	VIRT	RES	SHR	S	%CPU	%MEM	TIME+	COMMAND
717	root	20	0	0	0	0	S	0,3	0,0	0:01.29	kworker/0:0
930	root	20	0	42688	3468	2932	R	0,3	0,3	0:00.03	top
1	root	20	0	138828	6732	5304	S	0,0	0,7	0:01.13	systemd
2	root	20	0	0	0	0	S	0,0	0,0	0:00.00	kthreadd
3	root	20	0	0	0	0	S	0,0	0,0	0:00.59	ksoftirqd/0
5	root	0	-20	0	0	0	S	0,0	0,0	0:00.00	kworker/0:0H
6	root	20	0	0	0	0	S	0,0	0,0	0:00.10	kworker/u2:0
7	root	20	0	0	0	0	S	0,0	0,0	0:00.89	rcu_sched
8	root	20	0	0	0	0	S	0,0	0,0	0:00.00	rcu_bh
9	root	rt	0	0	0	0	S	0,0	0,0	0:00.00	migration/0
10	root	0	-20	0	0	0	S	0,0	0,0	0:00.00	lru-add-drain
11	root	rt	0	0	0	0	S	0,0	0,0	0:00.00	watchdog/0

Comando top (continuación)

- Fila 1:
 - **top** es el nombre del programa.
 - **20:55:23** es la hora actual.
 - **up 57 min** indica que el equipo ha estado encendido por 57 minutos.
 - **load average: 0.00, 0.00, 0.00** muestra la cantidad de procesos (en promedio) que están utilizando (o han estado esperando para utilizar) el CPU durante los últimos 60 segundos, 5 minutos, y 15 minutos, respectivamente.
- Fila 2:
 - **Tasks: 161 total, 1 running, 160 sleeping, 0 stopped, 0 zombie** muestra la cantidad de procesos que están corriendo actualmente en el sistema y sus posibles estados:
 - **running** indica la cantidad de procesos que están corriendo. En la sección indicada con el 2 se los identifica con la letra R en la columna S (de Status).
 - **sleeping** representa el número de procesos que no están corriendo actualmente pero que se encuentran esperando que ocurra un evento para despertarse (por ejemplo, una consulta al servidor web). Se indican con la letra S en la misma columna mencionada anteriormente.
 - **stopped** son aquellos procesos que han sido detenidos. Se identifican con la letra T.
 - Los procesos **zombie** se indican con la letra Z

Comando top (continuación)

- **PID:** Process ID.
- **USER:** Usuario dueño del proceso.
- **PR:** Prioridad de ejecución del proceso.
- **NI:** Es un valor que refleja la prioridad sobre el uso de los recursos del sistema otorgada a cada proceso en particular.
- **VIRT:** Cantidad de memoria virtual que el proceso está manejando.
- **RES:** Es la representación más cercana a la cantidad de memoria física que un proceso está utilizando.
- **SHR:** La cantidad de memoria compartida (potencialmente con otros procesos) disponible para este proceso en particular, expresada en KiB.
- **S:** Estado del proceso.
- **%CPU:** Es el porcentaje de tiempo de CPU utilizado por el proceso desde el último refresco de pantalla.
- **%MEM:** Indica el porcentaje correspondiente al uso de memoria física de un proceso en particular.
- **TIME+:** Tiempo de CPU que ha utilizado el proceso desde que inició, expresado en minutos:segundos.centésimas de segundo.
- **COMMAND:** Muestra el comando que se utilizó para iniciar el proceso o el nombre de este último.

Comando top (continuación)

- Como toda herramienta de la línea de comandos, top posee numerosas opciones. Entre las más útiles podemos destacar las siguientes:
 - Refrescar la pantalla cada X segundos:
top -d X
 - Ordenar la salida por uso de RAM (descendente):
top -o %MEM
 - Monitorear sólo los procesos cuyos PIDs son 324, 697, y 25216:
top -p 324,697,25216



Comando htop

- A diferencia de top, este comando es visualmente más atractivo.

```
0[|||||] 19.0% 4[|||||] 15.2%
1[|||||] 12.2% 5[|||||] 10.7%
2[|||||] 7.4% 6[|||||] 11.7%
3[|||||] 8.0% 7[|||||] 13.3%
Mem[|||||] 5.35G/21.4G Tasks: 202, 1006 thr, 167 kthr; 3 running
Swp[|||||] 0K/2.00G Load average: 0.73 0.64 0.73
Uptime: 04:54:03
```

PID	USER	PRI	NI	VIRT	RES	SHR	S	CPU%	MEM%	TIME+	Command
3652	gmart	20	0	1395G	545M	135M	S	14.6	2.5	4:07.55	/app/brave/brave --type=renderer --string-annotations --cr
2523	gmart	20	0	5139M	251M	136M	S	12.2	1.2	12:22.99	cinnamon --replace
333671	gmart	20	0	15500	6272	3456	R	12.2	0.0	0:01.69	htop
1423	root	20	0	1434M	204M	117M	S	8.5	0.9	16:05.25	/usr/lib/xorg/Xorg -core :0 -seat seat0 -auth /var/run/lig
2607	gmart	20	0	670M	13616	8360	S	2.4	0.1	5:16.68	conky -c /home/gmart/.conky/conkyrc1
2611	gmart	20	0	670M	13616	8360	S	2.4	0.1	6:04.38	conky -c /home/gmart/.conky/conkyrc1
2937	gmart	20	0	32.7G	508M	230M	S	2.4	2.3	5:59.87	/app/brave/brave --disable-features=WebAssemblyTrapHandler
115603	gmart	20	0	2126M	25792	16576	S	2.4	0.1	3:48.16	C:\windows\system32\winedevice.exe
1458	root	20	0	1434M	204M	117M	R	1.8	0.9	4:29.71	/usr/lib/xorg/Xorg -core :0 -seat seat0 -auth /var/run/lig
3058	gmart	20	0	32.7G	261M	145M	S	1.8	1.2	6:06.98	/app/brave/brave --type=gpu-process --string-annotations --
3083	gmart	20	0	32.4G	142M	104M	S	1.8	0.7	3:38.04	/app/brave/brave --type=utility --utility-sub-type=network
3661	gmart	20	0	1395G	545M	135M	S	1.8	2.5	0:35.91	/app/brave/brave --type=renderer --string-annotations --cr
302972	gmart	20	0	573M	72304	57988	S	1.8	0.3	0:02.32	/usr/bin/tilix --gapapplication-service

```
F1Help F2Setup F3Search F4Filter F5Tree F6SortBy F7Nice F8Nice + F9Kill F10Quit
```

Comando nice y renice

- Las columnas PR y NI en la salida de top indican la prioridad que se le ha asignado a un proceso en particular -tal como lo ve el kernel en un momento dado- y el valor de niceness del mismo.
- **PR y NI** están relacionados entre sí: mientras mayor sea la prioridad de un proceso, es menos nice (del inglés bueno) ya que consumirá más recursos del sistema.
- El valor de niceness puede ubicarse en algún lugar del intervalo comprendido entre -20 y 19. Estos dos límites representan la mayor y la menor prioridad posibles, respectivamente.
- Para modificar la prioridad de un proceso en ejecución, utilizaremos el comando renice. Los usuarios con privilegios limitados solamente pueden aumentar el valor de niceness correspondientes a un proceso del cual son dueños, mientras que root puede modificar este valor para cualquier proceso sin importar quién sea el usuario dueño del mismo.
- Cambiar la prioridad del proceso con PID 324 (opción -p) a 10: **renice -n 10 -p 324**
- Cambiar la prioridad de todos los procesos del usuario (opción -u) alumno a -10: **renice -n -10 -u alumno**
- **Por defecto**, cualquier nuevo proceso se ejecuta con una prioridad igual a 0. Si deseáramos iniciarlo con una prioridad diferente, podemos hacer uso del comando nice seguido de la opción -n, del nuevo valor de niceness deseado, y del comando a ejecutar.
- Por ejemplo, iniciemos top con un valor de niceness igual a 10: **nice -n 10 top**

Administración de procesos

- **systemctl** es una herramienta provista como parte de **systemd**:
 - **Iniciar el servicio:** **systemctl start miservicio.service** o **service miservicio start**
 - Configurarlos para que inicie al arrancar el equipo: **systemctl enable miservicio.service**
- Detenerlo: **systemctl stop miservicio.service** o **service miservicio stop**
- Impedir que inicie al arrancar el equipo: **systemctl disable miservicio.service**
- Reiniciar el servicio: **systemctl restart miservicio.service** o **service miservicio restart**
- Averiguar si está configurado para arrancar al inicio: **systemctl is-enabled miservicio.service**
- Averiguar si está corriendo actualmente: **systemctl is-active miservicio.service** o **systemctl -l status miservicio.service** (esta última variante provee más información sobre el estado y la operación del servicio).

```
gmart@informatica:~$ service cron status
● cron.service - Regular background program processing daemon
   Loaded: loaded (/usr/lib/systemd/system/cron.service; enabled; preset: enabled)
   Active: active (running) since Tue 2025-02-18 08:04:49 -03; 5h 15min ago
     Docs: man:cron(8)
    Main PID: 957 (cron)
      Tasks: 1 (limit: 26173)
     Memory: 3.4M (peak: 9.1M)
        CPU: 1.649s
    CGroup: /system.slice/cron.service
            └─957 /usr/sbin/cron -f -P
```

Procesos en Primer y Segundo plano

- En algunos casos, un script puede tomar un tiempo considerable en completar su ejecución, o un programa puede ocupar la línea de comandos mientras se encuentre corriendo. Para permitirnos volver a tomar el control de la terminal en cuestión, la shell nos permite colocar procesos en segundo plano. También podemos iniciar directamente un proceso en segundo plano, de manera que no ocupe la terminal mientras corre.
- Para que un proceso se inicie en segundo plano, colocaremos el símbolo **&** al fin del mismo.
- Al iniciar un proceso en segundo plano, la terminal queda libre para seguir trabajando en la misma y se nos provee la identificación de dicho proceso y su PID.
- Si deseamos traer un proceso a primer plano, nos valdremos el comando **fg** seguido del identificador de este. Si sucediera que no supiéramos el identificador del proceso en cuestión, podemos valernos del comando **jobs** para averiguarlo.

```
gmart@informatica:~$ vi 1.txt &
[1] 362712
gmart@informatica:~$ jobs
[1]+  Detenido                  vi 1.txt
gmart@informatica:~$ fg 1
```

Comando nohup

- La ejecución de un proceso se verá interrumpida si cerramos la terminal desde la que lo iniciamos, o si cerramos la sesión actual. Para evitar que esto suceda, podemos usar nohup, una herramienta que permite iniciar un proceso que sea inmune a la situación que describimos anteriormente.
- El comando **nohup** es especialmente útil cuando estamos conectados a un equipo de manera remota y deseamos ejecutar un script que continúe su ejecución luego de que cerremos la terminal asociada. De esta manera, no tenemos que estar logueados durante todo el tiempo que corra el script.
- La sintaxis del uso de esta herramienta es la siguiente:
nohup [programa a iniciar] &
- Como podemos ver, generalmente nohup se utiliza en conjunto con el símbolo **&** para enviar simultáneamente el programa en cuestión a segundo plano, aunque no es estrictamente necesario que lo hagamos de esa manera.
- Ejemplo: **nohup wget https://pagina.com.ar/ejemplo.txt &**

TMUX

- tmux es un multiplexor de terminal popular que permite ejecutar múltiples sesiones dentro de una sola ventana.
- Nos permite cerrar la terminal remota sin que se cierren las aplicaciones.
- **Instalación:** `sudo apt-get tmux`

The image is a screenshot of a tmux cheat sheet. It is organized into two main columns. The left column lists commands for creating, managing, and deleting sessions, along with keyboard shortcuts for switching between them. The right column lists commands for attaching to sessions, listing sessions, and navigating between windows and sessions. The background is dark with light text, and some commands are highlighted in green.

Sessions

`$ tmux`
`$ tmux new`
`$ tmux new-session`
`: new`
Start a new session

`$ tmux new-session -A -s mysession`
Start a new session or attach to an existing session named mysession

`$ tmux new -s mysession`
`: new -s mysession`
Start a new session with the name mysession

`: kill-session`
killDelete the current session

`$ tmux kill-ses -t mysession`
`$ tmux kill-session -t mysession`
killDelete session mysession

`$ tmux kill-session -a`
killDelete all sessions but the current

`$ tmux kill-session -a -t mysession`
killDelete all sessions but mysession

`Ctrl + b $`
Rename session

`Ctrl + b d`

`: attach -d`
Detach others on the session (Maximize window by detach other clients)

`$ tmux ls`
`$ tmux list-sessions`
`Ctrl + b s`
Show all sessions

`$ tmux a`
`$ tmux at`
`$ tmux attach`
`$ tmux attach-session`
Attach to last session

`$ tmux a -t mysession`
`$ tmux at -t mysession`
`$ tmux attach -t mysession`
`$ tmux attach-session -t mysession`
Attach to a session with the name mysession

`Ctrl + b w`
Session and Window Preview

`Ctrl + b (`
Move to previous session

`Ctrl + b)`
Move to next session

Comandos para apagar Linux

- El comando shutdown se utiliza para apagar o reiniciar Linux desde la terminal. Sintaxis: ***shutdown [OPCIÓN] [TIEMPO] [MENSAJE]***
- ***shutdown -h*** o ***shutdown*** #Linux se apaga en un minuto
- ***shutdown -h 0*** o ***shutdown now*** # linux se apaga inmediatamente
- ***shutdown -h 20*** o ***shutdown +20*** # se apaga en 20 minutos
- ***shutdown -h 17:30*** o ***shutdown 17:30*** # se apaga a las 17:30
- ***shutdown -c*** # cancela el pagado
- ***shutdown 'ESCRIBIR AQUÍ EL MENSAJE DE PARED'*** # El mensaje aparecerá en la pantalla de los usuarios del sistema operativo.
- ***init 0***



Comandos para reiniciar Linux

- El comando `shutdown` se utiliza para apagar o reiniciar Linux desde la terminal.
Sintaxis: **`shutdown [OPCIÓN] [TIEMPO] [MENSAJE]`**
- **`shutdown -r`** #Linux se reinicia en un minuto
- **`shutdown -r 0`** o **`shutdown -r now`** # linux se reinicia inmediatamente
- **`shutdown -r 20`** o **`shutdown -r +20`** # se reinicia en 20 minutos
- **`shutdown -r 17:30`** # se reinicia a las 17:30
- **`shutdown -c`** # cancela el reinicio
- `init 6`



¡Ahora tú!

¡PRACTICA!

ES LA MEJOR FORMA DE
APRENDER

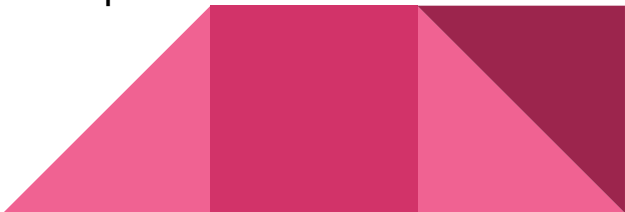


Práctica

Comando PS: Muestra los procesos

- `ps -a` muestra todos los procesos
- `ps -a -o comm,tty,pid,pri`
- `ps -aux | more` muestra todos los procesos aunque no tengan terminal asociado
- `ps -aux | grep -e "firefox"` busco por ejemplo los procesos llamados firefox
- `pstree` muestra al arbol de procesos -p muestra los pid
- cuando se crea un proceso se crea una carpeta en `/proc` con el nombre de su ID

Comando Nice: Asigna prioridad a los procesos. A mayor valor negativo más prioridad a mas positivo menos prioridad. La prioridad por defecto de nice es 10

- Nano prioridad1.txt creo archivo prioridad1.txt y lo abro con el editor de texto
 - `nice -n 6 nano prioridad1.txt`
 - `renice 4 -p 2234` (solo permite bajar la prioridad) solo root puede dar prioridad más alta
- 

Práctica

Comando TOP: Muestra los procesos en tiempo real

- `top -d 1` muestra los procesos en tiempo real
- `top -p id` muestra el estado del proceso

Comando Job: lista las tareas que se están ejecutando

- `job -l`

Comando Kill: envía señales a los procesos

- `kill -L` muestra opciones
 - `kill id` mata procesos
 - `kill -9 id` fuerza el cierre del proceso
 - `killall firefox` mata todos los procesos de por ejemplo firefox
- 